

Version 6

CC-102

WorkText in Programming 2

Rodibelle F. Leona

UNIT 1. LOOPS



Learning Objectives

At the end of the unit, the students are able to:

1. identify the loop or repetition structure;
 2. identify the different elements of the loop;
 3. differentiate the increment and decrement operators; and
 4. create simple and complicated programs using loops.
-

When writing C++ codes, there are times that you need to do a statement or a block of statements more than once, twice, 5 times, 10 times or even hundred of times. For example, you are asked to display your name and age on the screen 5 times, the following code will be part of your C++ program.

```
cout<<"Matty T. Laucan 18"<<endl;  
cout<<"Matty T. Laucan 18"<<endl;  
cout<<"Matty T. Laucan 18"<<endl;  
cout<<"Matty T. Laucan 18"<<endl;  
cout<<"Matty T. Laucan 18"<<endl;
```

It is okay if the specific number of times is less than 10, but how about if it is more than 10 times or a hundred times or a thousand times? This will be a tedious job even for a seasoned programmer. C++ provides a structure which will help you do repetitive tasks without typing them over and over again. It is called the **repetition structure** or **looping**. Another word for looping is **iteration**.

ELEMENTS OF LOOPING

A **loop** will not function properly if one of its important elements is missing.

1. **Loop Control Variable (LCV)**

The **loop control variable** will hold the value that will be compared to the limit of the loop. This limit will determine when the

loop will end. The loop control variable should first be initialized before it can be used in the program.

2. **Sentinel Value (SV)**

The **sentinel value** is the value where the loop control variable will be compared to. This value will decide if the loop will continue or stop based on the result of the comparison. The result of the comparison is always a Boolean value, that is, **true** or **false**.

3. **Loop update (LU)**

Inside the loop, the value of the loop control variable must be altered. This is called the **loop update**. You can change this value by incrementing or decrementing.

The **increment operator (++)** adds 1 to its operand, and the **decrement operator (--)** subtracts 1 from its operand. Thus:

x++;

which is the same as

x = x + 1;

will evaluate to **5** if the initial value of **x** is **4**.

Similarly,

x--;

which is the same as

x = x - 1;

will evaluate to **10** if the initial value of **x** is **11**.

Both the increment and decrement operators can either **precede (prefix)** or **follow (postfix)** the **loop control variable**.

For example:

x = x + 1;

can be written as

`++x;`

or as

`x++;`

The first is the prefix form and the second one is the postfix form of the statement **`x = x + 1;`**

When an increment or decrement is used as part of an expression, there is an important difference in using the prefix and postfix forms. If you are using prefix form, the increment or decrement will be done before evaluating the expression. And if you are using postfix form, the increment or decrement will be done after the complete expression is evaluated.

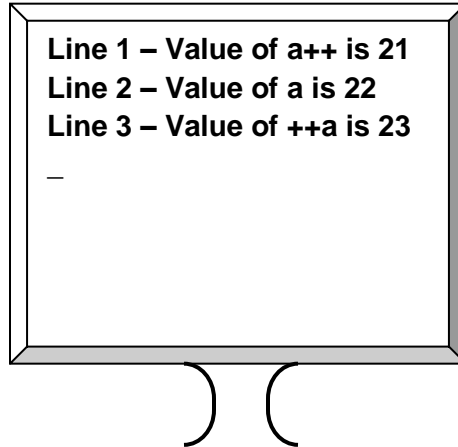
SAMPLE PROBLEM

Create a C++ program that will display the postfix and prefix increment values of a variable whose initial value is 21.

SAMPLE PROGRAM

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int main()
7 {
8     int a = 21;
9     int c ;
10
11     c = a++;
12
13     cout << "Line 1 - Value of a++ is " << c << endl ;
14     cout << "Line 2 - Value of a is " << a << endl ;
15
16     c = ++a;
17
18     cout << "Line 3 - Value of ++a is " << c << endl ;
19
20     return EXIT_SUCCESS;
21 }
```

SAMPLE OUTPUT



Increment and decrement are not only for adding or subtracting **1** from the current value of the loop control variable. They can be **any value** you wish to add or subtract from the value of the loop control variable. The following are examples of the increment and decrement by other numerals aside from **1**.

```
int x = 5;
```

UPDATE	CURRENT VALUE OF x
<code>x = x + 2;</code>	7
<code>x -= 5;</code>	2
<code>x -= 9;</code>	-7
<code>x += 2;</code>	-5
<code>x += 10;</code>	5
<code>x -= 6;</code>	-1
<code>x = x - 12;</code>	-13
<code>x += 7;</code>	-6
<code>x -= 14;</code>	-20

If you do not alter the loop control variable, it will result to an **infinite loop**.

An **infinite loop** is a loop which never stops. You have to be careful so that you will not end up with an infinite loop.

There are also loops that will not do anything which are called **empty loops**. This happens when the **comparison expression** in a **for loop** or a **while loop** evaluates to **false** even on the first try. You have to make sure that the comparison expression will evaluate to **true** at least **once**. What will be the good of using a loop if at the first time that you compare the value of the loop control variable to the sentinel value, it evaluates to false? The loop body will not be executed and it will proceed to the statement right after the loop. In this case, the loop structure will be like a design in your program and does not do anything. There will be no sense in including this type of loop in your program.

SAMPLE PROGRAM

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    int x;

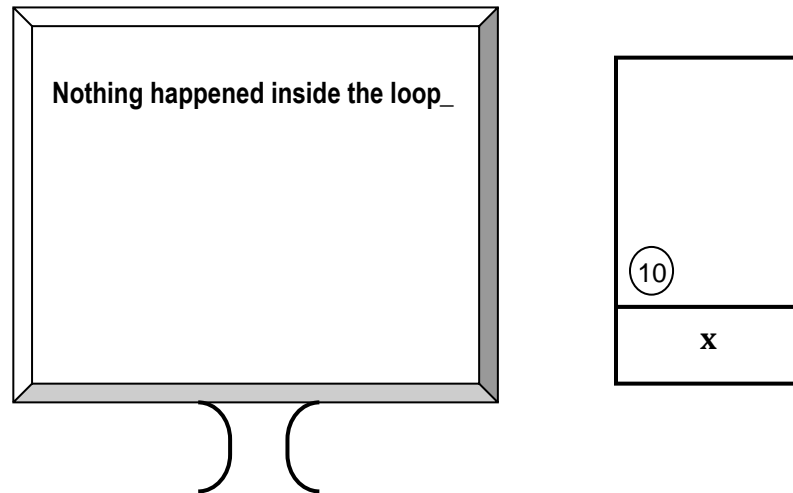
    x = 10;
    while (x <= 6)
    {
        cout<<"CHOOBAH ";

        x--;
    }

    cout<<endl;
    cout<<"Nothing happened inside the loop";

    return EXIT_SUCCESS;
}
```

SAMPLE OUTPUT



TYPES of LOOPS

1. *for* LOOP
2. *while* LOOP
3. *do while* LOOP
4. *nested* LOOP

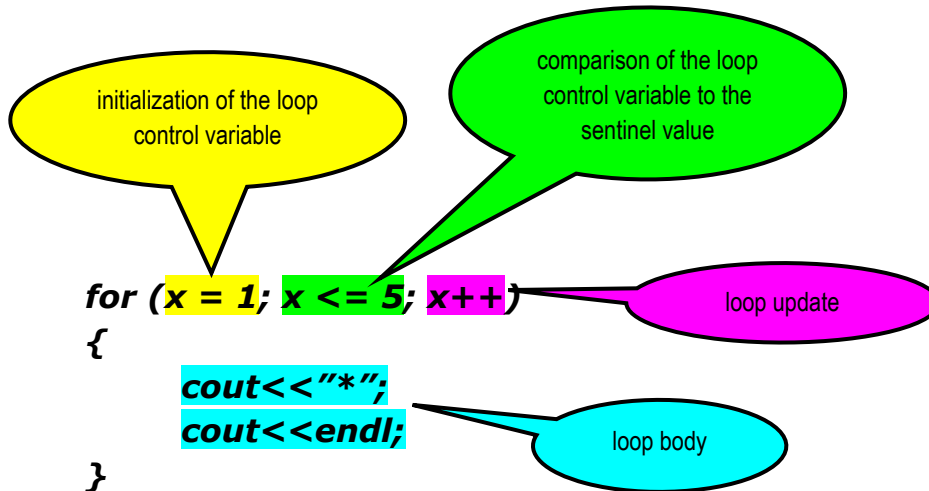
THE *for* LOOP

A *for* loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

The syntax of a for loop in C++ is:

```
for (initialization; comparison expression; update)  
{  
    statement(s);  
}
```

EXAMPLE



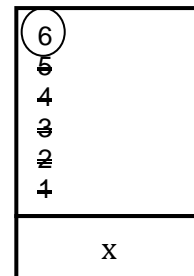
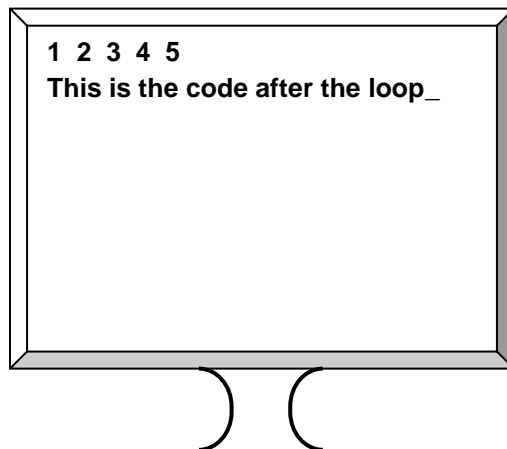
FLOW OF CONTROL IN A *for* LOOP

1. The **initialization of the loop control variable** is executed first, and only once. In here, the loop control variable is assigned with its initial value.
2. The **comparison expression** is evaluated next. If it evaluates to **true**, the body of the loop is executed. If it is **false**, the body of the loop does not execute and the flow of control jumps to the next statement just after the closing brace (`}`) of the **for** loop.
3. After the **body of the for loop** is executed, the flow of control jumps back up to the **update** statement. This statement allows you to update the loop control variable. The update can be an **increment** or a **decrement** depending on the initial value of the loop control variable, the comparison operator used and the sentinel value.

SAMPLE PROGRAM

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int main()
7 {
8     int x;
9
10    for (x = 1; x <= 5; x++)
11    {
12        cout<<x<<" ";
13    }
14
15    cout<<endl;
16    cout<<"This is the code after the loop";
17
18    return EXIT_SUCCESS;
19 }
```

SAMPLE OUTPUT



SIMULATION

- **Line number 1** invokes the iostream header file which holds the input/output commands of C++.

- **Line number 2** invokes the *cstdlib* library which will allow you to do math operations, random number generation, dynamic memory management and other C++ functions.
- **Line number 4** allows the use of standard commands like *cout*, *endl* without repeatedly typing *std::*.
- **Line number 6** starts the main method of the program.
- On **line number 8**, the computer will allocate memory space for variable **x** where the values that you will store should only be **integer** values.
- **Line number 10** sets the initial value of **x** to **1** (**x = 1**), then compare the value of **x** if it is less than or equal to 5 (**x <= 5**). Since the comparison evaluates to **true**, the code inside the loop body on **line number 12** will be executed (*cout<<x<<" "*). The value of **x** which is **1** will be printed on the screen followed by a **space**.
- Next, **x** becomes **2** since (**x++**) will add **1** to the current value of **x**.
- The simulation goes back to the **comparison expression** where the computer will evaluate the expression, **2 <= 5** which is **true**. It will then execute the loop body and print the current value of **x**, which is **2**, on the screen followed by a **space**.
- Next, **x** becomes **3**.
- Comparing **3 <= 5** evaluates to **true**, so **3** will be printed followed by a **space**.
- Next, **x** becomes **4**.
- Comparing **4 <= 5** evaluates to **true**, so **4** will be printed followed by a **space**.
- Next, **x** becomes **5**.
- Comparing **5 <= 5** evaluates to **true**, so **5** will be printed followed by a **space**.
- Next, **x** becomes **6**.
- Comparing **6 <= 5** evaluates to **false**. In this instance, the execution of the loop body will **stop** and the computer will execute the next lines following the **for** loop.
- The computer will now print a **newline** (*cout<<endl*) found in **line number 15**. Then, it will proceed to **line number 16** which prints the **message** (*cout<<"This is the code after the loop"*).
- **Line number 18** will print a message that you have successfully executed the program.
- **Line number 19** ends the main method of the program.

Seat No.: _____ Rating: _____
Name: _____ Date: _____

Seatwork No. 1

I. EVALUATE THE VALUE OF THE FOLLOWING UPDATES.

CODE	VALUE OF x
x = 11;	
x += 13;	
x = x - 23;	
x -= 32;	
x++;	
x -= 2;	
x--;	
x = x + 6;	
x += 17;	
x -= 4;	

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 2

SIMULATE THE FOLLOWING PROGRAM. USE THE SCREEN FOR THE OUTPUT AND THE BOXES FOR THE MEMORY ALLOCATIONS OF THE VARIABLES NEEDED. USE ONLY THE EXACT NUMBER OF BOXES FOR THE DECLARED VARIABLES AND DISREGARD THE REST OF THE BOXES.

```
#include <iostream>
#include <cstdlib>

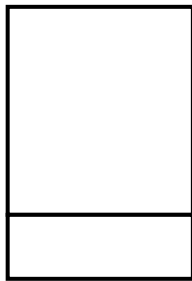
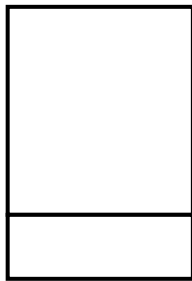
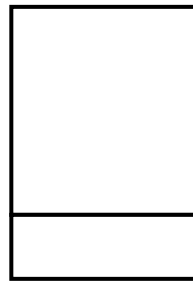
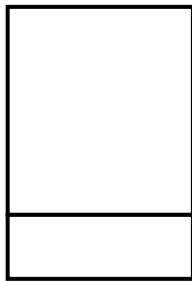
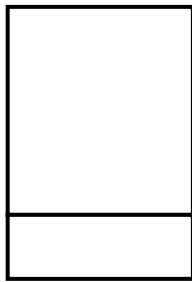
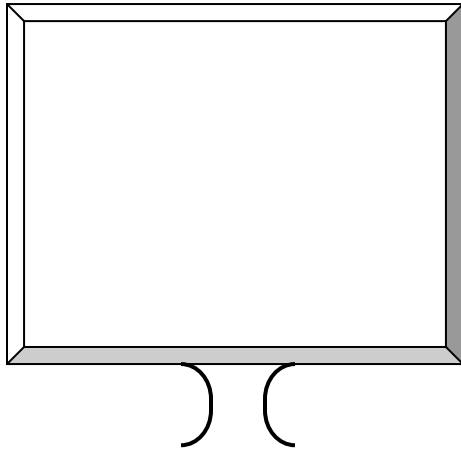
using namespace std;

int main()
{
    int x;

    for (x = 10; x >= 6; x--)
    {
        cout<<x<<" ";
    }

    cout<<endl;
    cout<<"FINISH";

    return EXIT_SUCCESS;
}
```



Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 3

SIMULATE THE FOLLOWING PROGRAM. USE THE SCREEN FOR THE OUTPUT AND THE BOXES FOR THE MEMORY ALLOCATIONS OF THE VARIABLES NEEDED. USE ONLY THE EXACT NUMBER OF BOXES FOR THE DECLARED VARIABLES.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    int x, square;

    for (x = 2; x <=10; x += 3)
    {
        square = x * x;

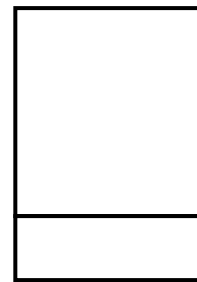
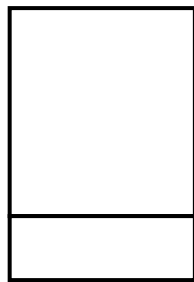
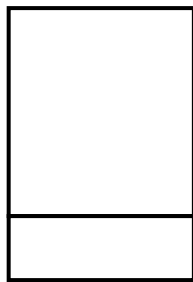
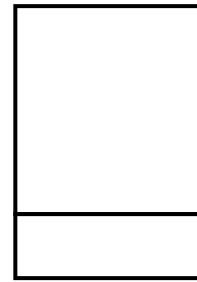
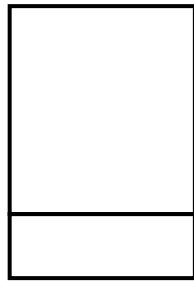
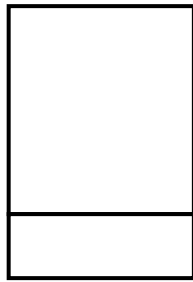
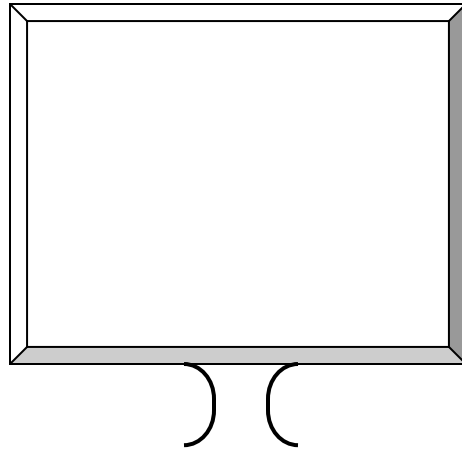
        cout<<square;
    }

    cout<<endl;

    square = 436;

    cout<<square<<endl<<endl;

    return EXIT_SUCCESS;
}
```



Seat No.: _____
Name: _____

Rating: _____
Date: _____

Laboratory Exercise No. 2

MAKE A C++ PROGRAM THAT WILL PRINT THE OUTPUT ON THE SCREEN USING *for* LOOP.

```
#1 #4 #9 #16 #25
```

) (

HINT:
The numbers are
square values.

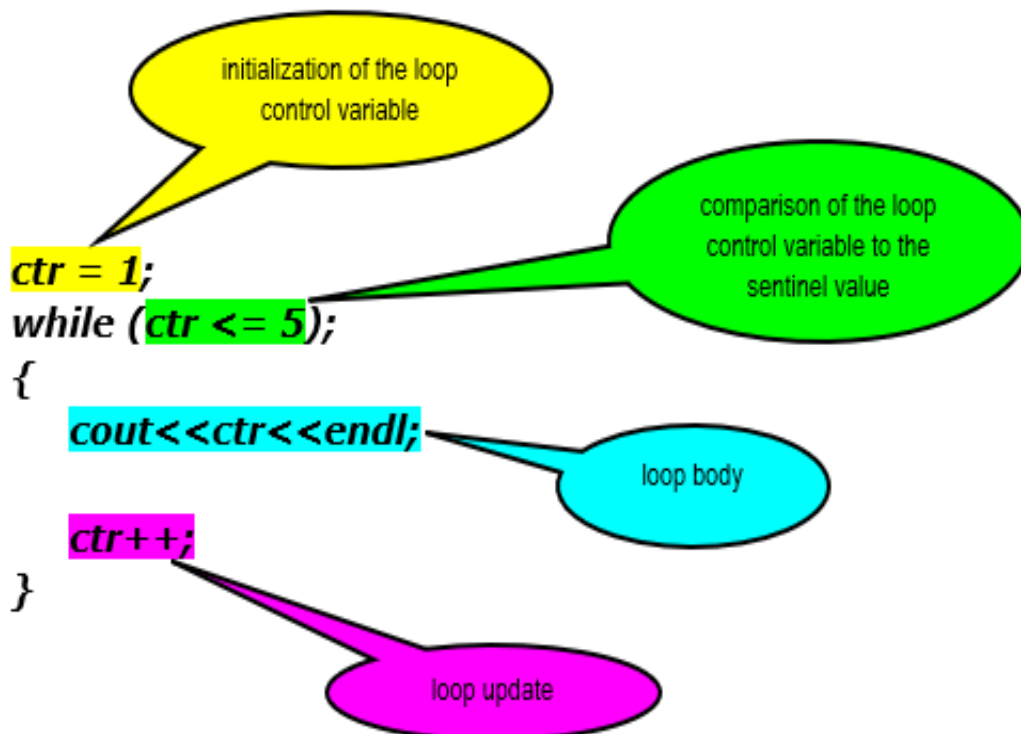
PROGRAM CODE

THE *while* LOOP

Another form of loop is the ***while*** loop. Basically, it will do the repetitive tasks assigned to it just like in a ***for*** loop. However, in a ***while*** loop, the initialization of the ***loop control variable*** is typed before the structure of the loop. The ***loop update*** on the other hand is placed inside the loop body. The syntax of the while loop is given below.

```
initialization;  
while (comparison expression)  
{  
    statement/s;  
    loop update;  
}
```

EXAMPLE:



SAMPLE PROGRAM

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    int x;

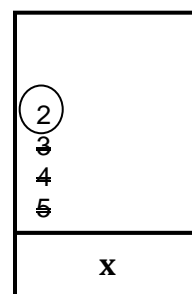
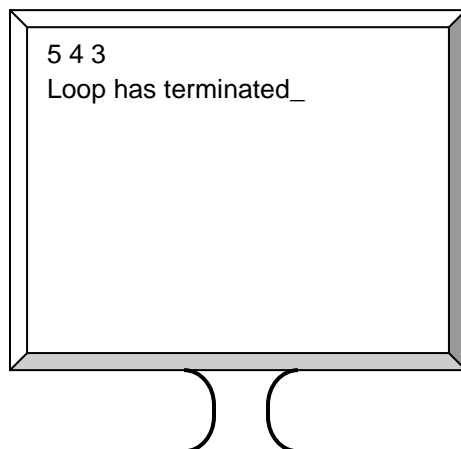
    x = 5;
    while (x >= 3)
    {
        cout<<x<<" ";

        x--;
    }

    cout<<endl;
    cout<<"Loop has terminated";

    return EXIT_SUCCESS;
}
```

SAMPLE OUTPUT



SIMULATION

- **Line number 6** starts the main method of the program.
- On **line number 8**, the computer will allocate memory space for variable **x** where the values that you will store should only be **integer** values.
- **Line number 10** sets the initial value of **x** to **5** (**x = 5**).
- On **line number 11**, the value of **x** is compared if it is greater than or equal to **3** (**x >= 3**). It will ask if **5 >= 3**. Since the comparison evaluates to **true**, the code inside the loop body on **line number 13** will be executed (**cout<<x<<" "**). The value of **x**, which is **5**, will be printed on the screen followed by a **space**.
- Next, **line number 15** will update the value of **x** and it will become **4** since (**x--**) will subtract **1** from the current value of **x**.
- The simulation goes back to the **comparison expression** where the computer will evaluate the expression, **4 >= 3** which is **true**. It will then execute the loop body and print the current value of **x**, which is **4**, on the screen followed by a **space**.
- Next, **x** becomes **3**.
- Comparing **3 >= 3** also evaluates to **true**, so **3**, which is the current value of **x**, will be printed followed by a **space**.
- Next, **x** becomes **2**.
- Comparing **2 >= 3** evaluates to **false**. In this instance, the execution of the loop body will **stop** and the computer will execute the next lines following the **for** loop.
- The computer will now print a **newline** (**cout<<endl**) found in **line number 18**. Then, it will proceed to **line number 19** which prints the **message** (**cout<<"Loop has terminated"**).
- **Line number 21** will print a message that you have successfully executed the program.
- **Line number 22** ends the main method of the program.

LOOP UPDATE USING A PROMPT

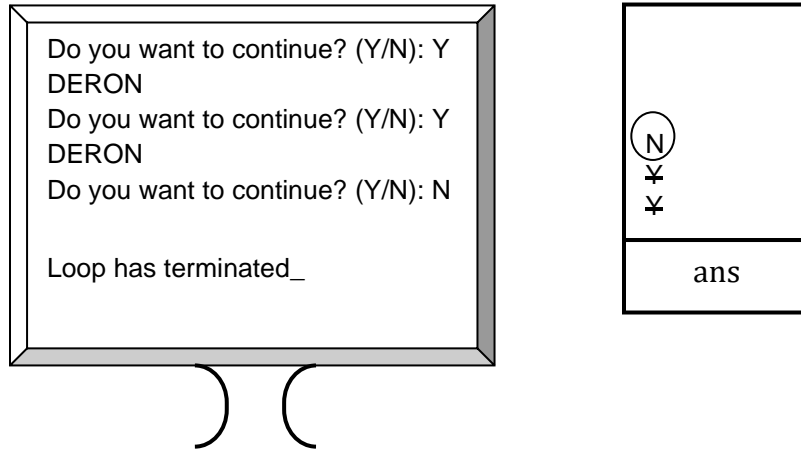
There are also instances when the loop has no definite number of times to execute, meaning, it will continue to do the iteration until such time that the user of the program tells the program to stop the execution of the looping process.

The loop update will be a question prompting the user if he wants to continue or stop executing the loop. So, every time the user enters a positive answer to the question, the loop body will be executed. It will stop the execution of the loop body when it receives a negative response to the question.

SAMPLE PROGRAM

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int main()
7 {
8     char ans;
9
10    cout<<"Do you want to continue? (Y/N): ";
11    cin>>ans;
12
13    while (ans == 'Y')
14    {
15        cout<<"DERON"<<endl;
16        cout<<"Do you want to continue? (Y/N): ";
17        cin>>ans;
18    }
19
20    cout<<endl;
21    cout<<"Loop has terminated";
22
23    return EXIT_SUCCESS;
24 }
```

SAMPLE OUTPUT



SIMULATION

- **Line number 6** starts the main method of the program.
- On **line number 8**, the computer will allocate memory space for variable **ans** where the values that you will store should only be **character** values.
- **Line number 10** prints the prompt on the screen (**cout<<"Do you want to continue? (Y/N): ";**).
- On **line number 11**, the value entered in the prompt by the user will be placed in variable **ans**.
- **Line number 13** (**while (ans == 'Y')**) will compare if the value entered by the user is **Y**.
- Since the value entered is **Y**, the loop body was executed. **DERON** was printed on the screen followed by a newline, as the statement (**cout<<"DERON"<<endl;**) instructed on **line number 15**.
- **Line number 17** (**cout<<"Do you want to continue? (Y/N): ";**) will print the prompt on the screen again and wait for the user to enter a value.
- Variable **ans** will get the value entered in the prompt as instructed on **line number 18** (**cin>>ans;**) which is the character **Y**. This will serve as the update of the loop.
- The execution of the code will go back to **line number 13** and compare if **Y** is equal to **Y**. Since the expression evaluates to **true**, the loop body will again be executed and a new value for **ans** will be set and the value of **ans** will again be compared to **Y**.

- Since the new value of **ans** is **N**, the loop terminates and the next line following the loop will be executed. **Line number 21** (`cout<<endl;`) will print a **new line** on the screen.
- Next will be **line number 22** (`cout<<"Loop has terminated";`) which will print the literal string on the screen.
- **Line number 24** will print a message that you have successfully executed the program.
- **Line number 25** ends the main method of the program.

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 4

MAKE A C++ PROGRAM USING while LOOP TO FIND THE FACTORIAL OF A POSITIVE INTEGER ENTERED BY USER. (Factorial of $n = 1*2*3...*n$).

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Laboratory Exercise No. 3

MAKE A C++ PROGRAM THAT WILL PRINT THE OUTPUT ON THE SCREEN USING while LOOP.

```
1
16
49
100
169
AY AD TRI END ISKWEYRD DA SAM.
```

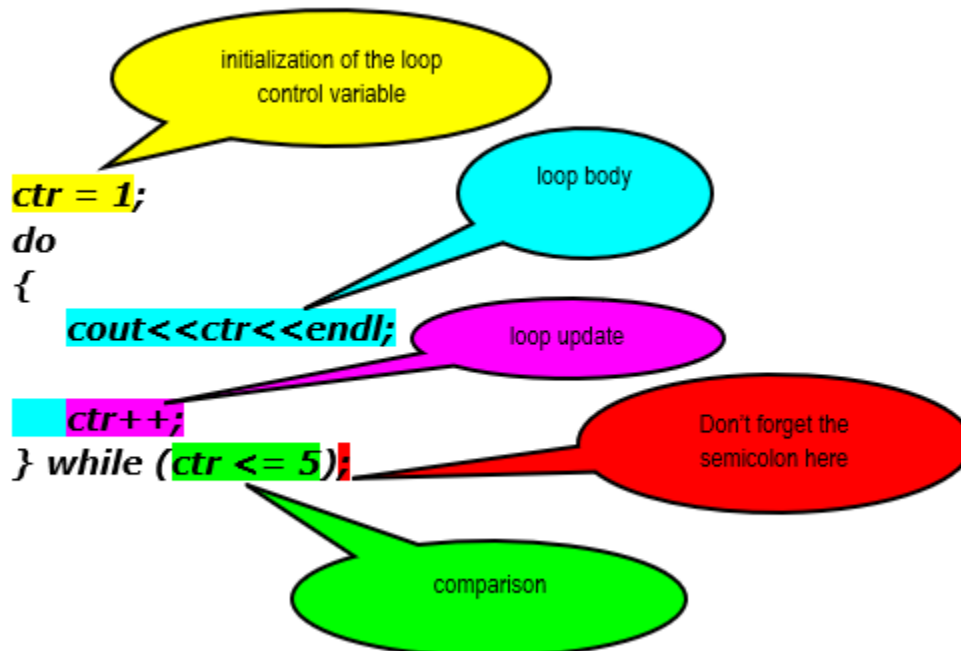
PROGRAM CODE

THE *do while* LOOP

The *do while loop* functions just like a *for* loop or a *while* loop. The only difference is that the comparison happens at the end of the *do while* loop so the body of the loop is executed **at least once** even if the comparison expression evaluates to *false*, unlike in the *for* loop and the *while* loop where the comparison expression happens before the loop body. This is the reason why, in the *for* and *while* loops, when the comparison evaluates to *false*, the loop body will not be executed even once. The structure of a *do while* loop is shown below.

```
initialization;  
do  
{  
    statement/s;  
    loop update;  
} while (comparison expression);
```

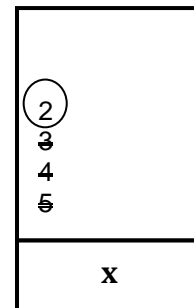
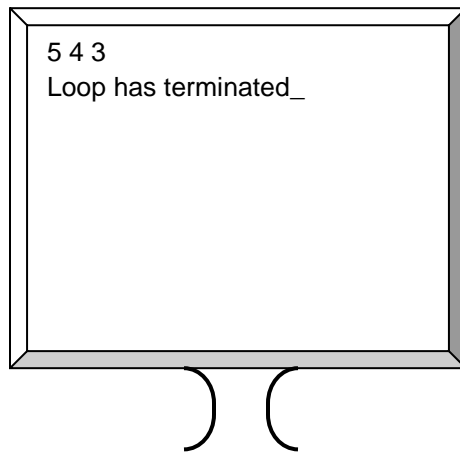
EXAMPLE



SAMPLE PROGRAM

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int main()
7 {
8     int x;
9
10    x = 5;
11    do
12    {
13        cout<<x<<" ";
14
15        x--;
16    } while (x >= 3);
17
18    cout<<endl;
19    cout<<"Loop has terminated";
20
21    return EXIT_SUCCESS;
22 }
```

SAMPLE OUTPUT



SIMULATION

- **Line number 6** starts the main method of the program.
- On **line number 8**, the computer will allocate memory space for variable **x** where the values that you will place should only be **integer** values.
- **Line number 10** sets the initial value of **x** to **5** (**x = 5**). **Line number 11** will start the **do while** loop.
- **Line number 13** will be executed (**cout<<x<<" "**). The value of **x** (**5**) will be printed on the screen followed by a **space**.
- Next, **line number 15** will update the value of **x** and it will become **4** since (**x--**) will subtract **1** from the current value of **x**.
- Then, the **comparison expression** on **line number 16** will be executed, where the computer will evaluate the expression, **4 >= 3** which is **true**. It will then execute the loop body and print the current value of **x** (**4**) on the screen followed by a **space**.
- Next, **x** becomes **3**. Comparing **3 >= 3** evaluates to **true**, so, the current value of **x** (**3**) will be printed followed by a **space**. Next **x** becomes **2**.
- Comparing **2 >= 3** evaluates to **false**. In this instance, the execution of the loop body will **stop** and the computer will execute the next lines following the **for** loop.
- The computer will now print a **newline** (**cout<<endl**) found in **line number 18**. Then, it will proceed to **line number 19** which prints the message (**cout<<"Loop has terminated"**).
- **Line number 21** will print a message that you have successfully executed the program.
- **Line number 22** ends the main method of the program.

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 5

MAKE A C++ PROGRAM THAT WILL INPUT A NUMBER AND DISPLAY THE SUM OF THE NUMBERS FROM 1 TO THE INPUT NUMBER. BE GUIDED BY THE SAMPLE OUTPUT GIVEN. USE *do while* LOOP.

Enter a no: 6
The sum is 21.

)
(

PROGRAM CODE

HINT:

When you enter 6, your program should add $1+2+3+4+5+6$ which is equal to 21.

When you enter 3, your program should add $1+2+3$ which is equal to 6.

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Laboratory Exercise No. 4

MAKE A C++ PROGRAM THAT WILL PRINT THE OUTPUT ON THE SCREEN USING *do while* LOOP.

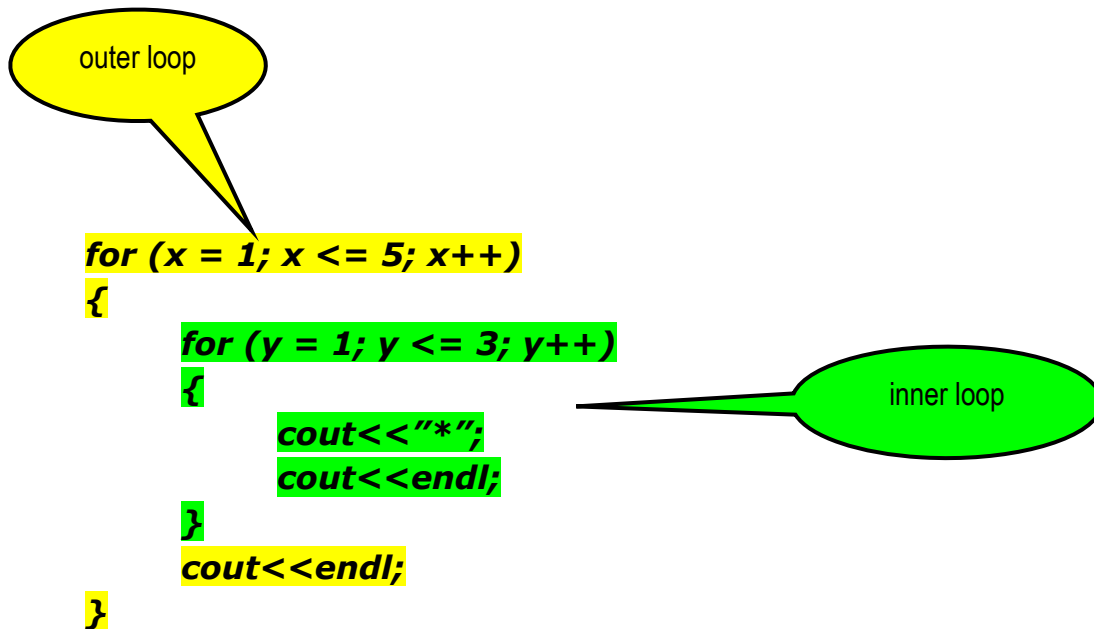
```
5
6
7
8
9
10
```

) (

PROGRAM CODE

NESTED LOOPS

A **nested loop** is a loop within a loop. It is consisting of an **outer loop** and an **inner loop**. The outer loop will be executed first. When the evaluation of the comparison operators evaluates to true, the loop body of the outer loop will be executed. Inside the body of the outer loop, the inner loop is embedded. The inner loop executes and when the comparison of the inner loop evaluates to **true**, the loop body of the inner loop will be executed repeatedly until such time that the comparison expression results to **false**. The inner loop terminates at this point and the rest of the statements inside the loop body of the outer loop will be executed. Then, the update of the outer loop will be performed and the process will be repeated until the outer loop terminates.



The number of times the **nested** loop will execute is equal to the number of times the **outer** loop executes times the number of times the **inner** loop executes.

$$tLoop = o * i;$$

Let's say that the **outer** loop will iterate **3 times (o)** and the **inner** loop will iterate **5 times (i)**, the **total iteration** will be **15**.

SAMPLE PROGRAM

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int main()
7 {
8     int x, y, loopCounter = 0;
9
10    for (x = 1; x <= 3; x++)
11    {
12        for (y = 1; y < 4; y++)
13        {
14            cout<<y<<" ";
15
16            loopCounter++;
17        }
18
19        cout<<endl;
20    }
21
22    cout<<endl;
23    cout<<"This loop iterates "<<loopCounter<<" times."<<endl<<endl;
24
25    return EXIT_SUCCESS;
26 }
```

SAMPLE OUTPUT

```
1 2 3
1 2 3
1 2 3

This loop iterates 9 times.

-
```

4 3 2 1
x

4 4 4
3 3 3
2 2 2
1 1 1
y

5
4
3 9
2 8
1 7
0 6
loopCounter

SIMULATION

- **Line number 6** starts the main method of the program.
- On **line number 8**, the computer will allocate memory spaces for variable **x**, variable **y** and variable **loopCounter** which will hold integer values only. It will also initialize **loopCounter** with a value of **0**.
- **Line number 10** sets the initial value of **x** to **1** (**x = 1**), then compare the value of **x** if it is less than or equal to 3 (**x <= 3**). The answer will be **true**.
- **Line number 12** will then be executed setting the initial value of **1** to variable **y**. Comparison comes next with (**x < 4**) which evaluates to **true**.
- The inner loop body will be executed and from **line number 14** (**cout<<y<<" "**), the value of **y** which is **1** will be printed on the screen followed by a **space**.
- **Line number 16** will add **1** to **loopCounter**, changing its value from **0** to **1**.
- Next, **y** will become **2** since (**y++**) in **line number 10** will add **1** to the current value of **x**.
- The simulation goes back to the **comparison expression** where the computer will evaluate the expression, **2 < 4** which is **true**. It will then execute the loop body and print **2** on the screen and a **space**. **1** will be again be added to the current value of **loopCounter** changing its value to **2**.
- Next, **y** becomes **3**.
- Comparing **3 < 4** evaluates to **true**, so **3** will be printed followed by a **space**. **1** will be again be added to the current value of **loopCounter** changing its value to **3**.
- Next **y** becomes **4**.
- Comparing **4 < 4** evaluates to **false**. In this instance, the execution of the inner loop body will **stop** and the computer will execute the next lines following the **for** loop.
- The computer will now print a **newline** (**cout<<endl**) found in **line number 19**.
- The outer loop body will finish on **line number 20** and it will increment variable **x**. So, from (**x++**) on **line number 10**, **x** will now have a value of **2**. It will then compare if **2** is less than or equal to **3** (**x <= 3**), which evaluates to **true**.

- **Line number 12** up to **Line number 20** will then be executed setting again the initial value of **1** to variable **y** which is in the inner loop. The comparison ($y < 4$) will evaluate to true and the body of the inner loop will be executed 3x (**Line number 13 to 17**) printing **1 2 3** on the screen and the value of **loopCounter** will be changed to 4, then 5, then 6. When the value of **y** becomes **4**, the inner loop will then stop its execution since the comparison ($y < 4$) evaluates to **false**.
- The computer will now print a **newline** (`cout<<endl`) found in **line number 19**.
- Variable **x** will now be updated again to 3 and the loop body of the outer loop will be executed again.
- **Line number 12** up to **Line number 20** will again be executed setting again the initial value of **1** to variable **y** which is in the inner loop. The body of the inner loop will be executed 3x (**Line number 13 to 17**) printing **1 2 3** on the screen and the value of **loopCounter** will be changed to 7, then 8, then 9. When the value of **y** becomes **4**, the inner loop will then stop its execution since the comparison ($y < 4$) evaluates to **false**.
- The computer will now print a **newline** (`cout<<endl`) found in **line number 19**.
- Next **x** becomes **4**.
- Comparing $4 \leq 3$ evaluates to **false**. In this instance, the execution of the outer loop will **stop** and the computer will execute the next lines following the **outer** loop.
- Then, it will proceed to **line number 22** which prints a **newline** (`cout<<endl`) and the **message** (`cout<<"This loop iterates "<<loopCounter<<" times"`) from **line number 23**.
- **Line number 25** will print a message that you have successfully executed the program.
- **Line number 26** ends the main method of the program.

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 6

MAKE A C++ PROGRAM THAT WILL DISPLAY THE SAMPLE OUTPUT USING nested LOOP.

```
11 12 13 14 15  
11 12 13 14  
11 12 13  
11 12  
11
```

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Laboratory Exercise No. 5

MAKE A C++ PROGRAM THAT WILL PRINT THE OUTPUT ON THE SCREEN USING do while LOOP. (NOTE: THERE ARE NO SPACES IN BETWEEN ROWS)

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

PROGRAM CODE

Seat No.: _____

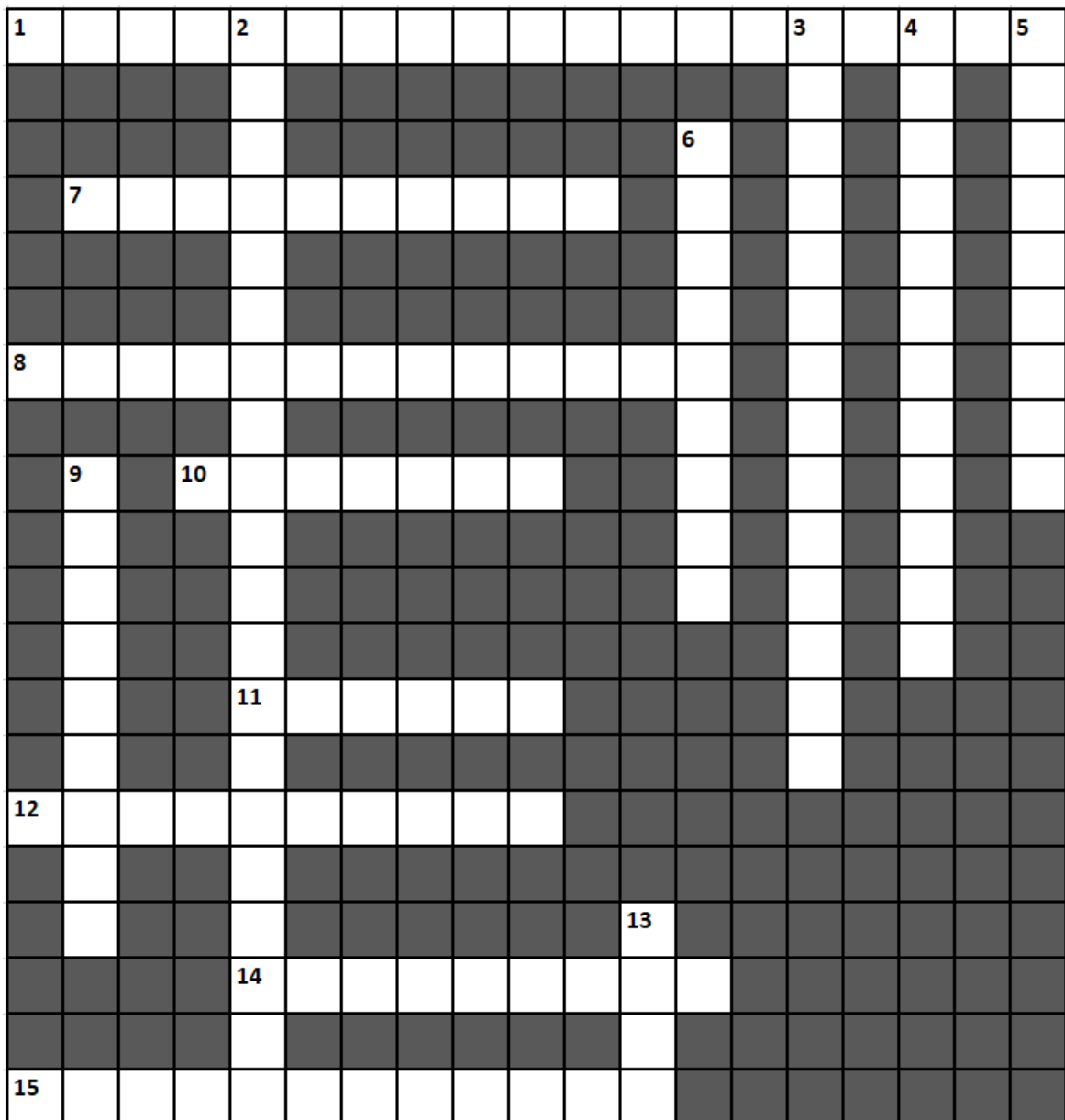
Rating: _____

Name: _____

Date: _____

CHAPTER TEST

- I. FILL-OUT THE CROSSWORD PUZZLE. THE NUMBER AFTER THE CLUES CORRESPONDS TO THE NUMBER OF WORDS THE ANSWER HAS.



ACROSS

1. Variable that controls the loop (3)
7. Increment or decrement (2)
8. The value where the lcv was compared to (2)
10. Increment or decrement after evaluation (1)
11. Increment or decrement before evaluation (1)
12. Loop within the loop (2)
14. Another term for looping or repetition structure (1)
15. A loop that never ends (2)

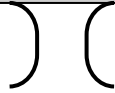
DOWN

2. Determines when the loop continues or ends (2)
3. Setting a first value to a variable (1)
4. True or false (2)
5. Loop that does nothing (2)
6. Increase the value of a variable (1)
9. Decrease the value of a variable (1)
13. A control structure that do a statement or a block of statements a certain number of times (1)

II. WHAT IS THE DIFFERENCE BETWEEN A *for* AND A *while* LOOP TO A *do while* LOOP.

III. WRITE A C++ PROGRAM THAT WILL PRINT THE OUTPUT GIVEN. USE ANY COMBINATION OF *nested* LOOPS.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```



HINT:

THE OUTPUT HAS SPACES BETWEEN NUMBERS AND THE PROGRAM MUST HAVE 2 *nested* LOOPS.

PROGRAM CODE

UNIT 2. ARRAYS



Learning Objectives

At the end of the unit, the students are able to:

- 5. identify the array structure;*
 - 6. differentiate the different types of arrays; and*
 - 7. create simple and complicated programs using arrays.*
-

An **array** is a series of elements with the same data type which are placed in adjoining memory locations. Each element of an array can be individually referenced by adding an index to the array's identifier.

Let's say that you are to use five quizzes of a student in a C++ program. In a regular C++ code, you have to declare each quiz with its corresponding variable name. So, you will have to declare five variable names for the five quizzes, for example:

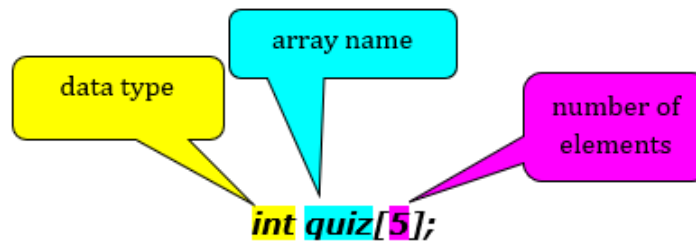
int quiz1, quiz2, quiz3, quiz4, quiz5;

Since the quizzes will be of the same type, say, ***int***, you can use an array to declare and use the quizzes. With an array declaration, you can place the quizzes in adjoining memory locations and each quiz can be accessed by calling the array name and its corresponding **index**. The index represents the memory location of an element in an array. The first element is always at ***index [0]***, therefore, the second will be at ***index [1]***, the third at ***index [2]*** and so on. The syntax for the declaration of an array is:

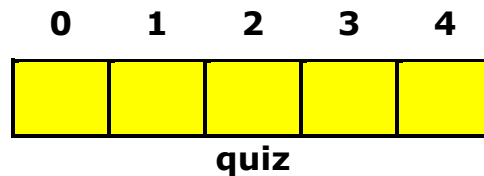
data_type array_name[no_of_elements];

where ***data_type*** is any valid data type such as ***int, float, short, char*** and others, ***array_name*** is a valid identifier and the ***number of elements*** enclosed in the open and close brackets signifies the ***length*** of the array. The number of elements must be a constant value since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

For an array of five quizzes, the declaration will be:



The compiler will allocate five adjoining memory locations for **array quiz**.



In C++, the first element in an array is always numbered with a 0 (not a 1), no matter its length. So since array **quiz** have **5** elements, the memory address starts with **quiz[0]** and ends with **quiz[4]**.

INITIALIZING ARRAYS

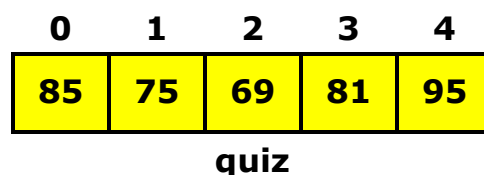
Usually, arrays are not initialized, meaning, they are not given values in the declaration part of the code. But you can initialize arrays by placing specific values when they are declared enclosed in opening and closing braces. The syntax for initializing an array is:

dataType arrayName[no_of_elements] = {val1, val2,..., valn};

EXAMPLE:

int quiz[5] = {85, 75, 69, 81, 95};

This initialization will cause the compiler to allot **5** adjoining memory locations with the following values.



The following shows the value of each element in the array.

quiz[0] = 85
quiz[1] = 75
quiz[2] = 69
quiz[3] = 81
quiz[4] = 95

The number of values inside the braces **MUST** not be greater than the length of array. If the number of values is less, the remaining elements will have the default value of 0.

int quiz[5] = {85, 75, 69};

This initialization will cause the computer to allot 5 adjoining memory locations with the following values.

0	1	2	3	4
85	75	69	0	0
quiz				

The value of each element in the array will be like this:

quiz[0] = 85
quiz[1] = 75
quiz[2] = 69
quiz[3] = 0
quiz[4] = 0

If the braces have no value inside, then the resulting memory locations allotted will look like this:

0	1	2	3	4
0	0	0	0	0
quiz				

and the value of each element in the array will be like this:

```
quiz[0] = 0  
quiz[1] = 0  
quiz[2] = 0  
quiz[3] = 0  
quiz[4] = 0
```

Also, the brackets of the array can be empty, meaning that the length of the array is not specified in the declaration. If this is the case, the compiler will assume the number of values in the array as its length. In this example:

```
int quiz[] = {85, 75, 69, 81};
```

the compiler will allot **4** memory locations for the array **quiz**.

0	1	2	3
85	75	69	81

quiz

ARRAY VALUES

Assigning the value of an element in an array is just like assigning the value of any regular variable. The syntax to do this is:

```
array_name[index_no];
```

For example,

```
choobah[3] = 32;
```

will store the value 32 in the fourth element of the array **choobah**.

0	1	2	3	4
			32	

choobah

Since, the index of an array always starts with **0** and is therefore the first element of the array, **choobah[1]** will be the second element, **choobah[2]** is the third element, **choobah[3]** is the fourth element and **choobah[4]** will be the fifth element.

You can also transfer/copy the value of an array element into a variable, let's say,

```
a = choobah[3];
```

therefore, variable **a** will have a value of **32**.

In C++, it is syntactically correct to exceed the valid range of indices for an array, meaning you will not encounter an error message when you compile your program. The problem will be on the output of the code. For example,

```
x = choobah[6];
```

This code will not create an error during compilation but the error will be seen during runtime.

At this point, it is important to be able to clearly distinguish between the two uses that **brackets []** have related to arrays. They perform two different tasks: one is **to specify the size of arrays when they are declared**;

```
int quiz[5];
```

and the second one is **to specify indices for concrete array elements when they are accessed**.

```
a = quiz[3];
```

Do not confuse these two possible uses of brackets [] with arrays.

SAMPLE PROBLEM

Make a C++ program that will add the values of an array with 5 elements.

SAMPLE PROGRAM

```
#include <iostream>
#include <cstdlib>

using namespace std;

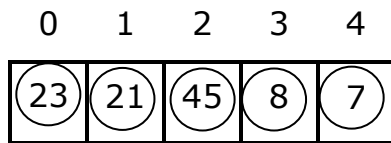
int main ()
{
    int num[] = {23, 21, 45, 8, 7};
    int x, sum=0;

    for ( x=0 ; x<5 ; x++ )
    {
        sum += num[x];
        cout<<sum<<endl;
    }

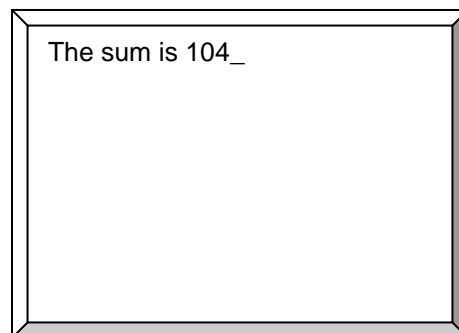
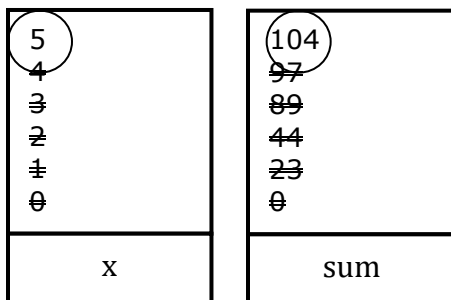
    cout <<"The sum is "<<sum;

    return EXIT_SUCCESS;
}
```

SAMPLE OUTPUT



num



Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 7

MAKE A C++ PROGRAM THAT WILL DISPLAY ALL THE CONTENTS OF AN INTEGER ARRAY WITH A LENGTH OF 5. USE A *for LOOP*. YOU SHOULD PROVIDE THE VALUES OF THE ARRAY.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 8

MAKE A C++ PROGRAM THAT WILL DISPLAY THE SUM OF THE VALUES OF AN INTEGER ARRAY WITH A LENGTH OF 6. THE VALUES OF THE ARRAY WILL BE INPUT VALUES FROM THE USER. USE *for* *LOOPS*.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Laboratory Exercise No. 6

MAKE A C++ PROGRAM THAT WILL PRINT THE AVERAGE OF THE VALUES OF AN ARRAY NAMED *areas* WITH VALUES {84, 76, 48} USING *while LOOP*.

PROGRAM CODE

MULTIDIMENSIONAL ARRAYS

A **multidimensional array** is an array of arrays. The most common of which is a **bi-dimensional array**, a two-dimensional array and can be imagined as a two-dimensional table made of elements with the same data types.

	0	1	2	3	4
0					
1					
2					

deron

This bi-dimensional array named **deron** is consists of **3** by **5** **elements** with data type **int**. To declare this array, the syntax is:

```
int deron[3][5];
```

To access the third element vertically and the second horizontally of this sample array, use the following code:

```
deron[2][1];
```

	0	1	2	3	4
0					
1					
2					

deron

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed.

SAMPLE PROGRAM

```
#include <iostream>
#include <cstdlib>

using namespace std;

#define WIDTH 5
#define HEIGHT 3

int main ()
{
    int jimmy [HEIGHT][WIDTH];
    int n, m, x, y;

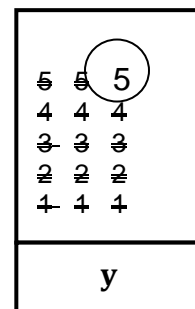
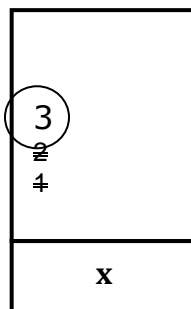
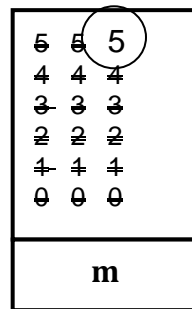
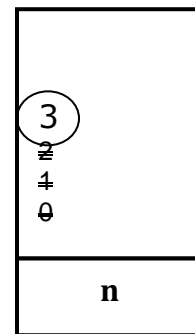
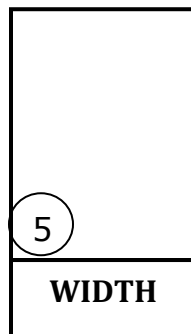
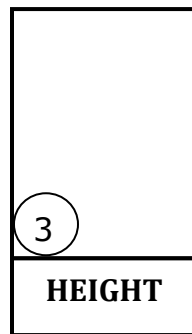
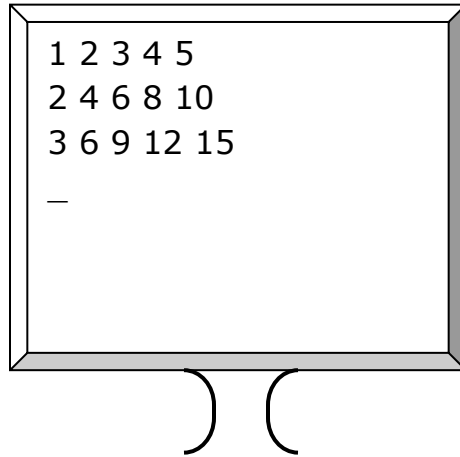
    for (n=0; n<HEIGHT; n++)
    {
        for (m=0; m<WIDTH; m++)
        {
            x = n + 1;
            y = m + 1;
            jimmy[n][m] = x * y;

            cout<<jimmy[n][m]<<" ";
        }

        cout<<endl;
    }

    return EXIT_SUCCESS;
}
```

SAMPLE OUTPUT



	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

jimmy

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 9

MAKE A C++ PROGRAM THAT WILL CREATE A MULTIPLICATION TABLE FROM 1 TO 10 USING MULTIDIMENSIONAL ARRAYS.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

CHAPTER TEST

I. IDENTIFY THE FOLLOWING.

1. It is a series of elements with the same data type which are placed in adjoining memory locations.

2. It is an array of arrays.

3. A two-dimensional array and can be imagined as a two-dimensional table made of elements with the same data types.

4. The number of elements enclosed in the open and close brackets signifies the _____ of the array.

5. It represents the memory location of an element in an array.

II. WRITE THE VALUE OF THE FOLLOWING. LABEL FIRST THE INDICES OF THE ARRAY.

1	12	85	15	42	95	11	77	52	56	17	32
18	64	78	82	31	59	88	0	3	86	72	45
99	25	75	16	33	54	19	10	44	38	66	69
84	55	47	29	30	20	90	80	70	35	74	22
65	23	61	4	7	8	6	15	37	57	67	60

bato

- bato[4][10] = _____
- bato[3][11] = _____
- bato[3][8] = _____
- bato[1][5] = _____
- bato[1][7] = _____
- bato[3][6] = _____
- bato[0][1] = _____
- bato[2][9] = _____
- bato[1][8] = _____
- bato[4][5] = _____

III. WRITE THE INDICES OF THE FOLLOWING VALUES.

- 0 = _____
- 25 = _____
- 38 = _____
- 47 = _____
- 52 = _____
- 60 = _____
- 78 = _____
- 84 = _____
- 95 = _____
- 17 = _____

UNIT 3. FUNCTIONS



Learning Objectives

At the end of the unit, the students are able to:

- 1. identify a function structure;*
 - 2. identify the different parts of a function structure; and*
 - 3. create simple and complicated programs using functions.*
-

Functions are modules or segments of code that perform individual tasks. In C++, a function is a group of statements that is given a name, and which can be called from some point in the program. The most common syntax to define a function is:

```
data_type function_name ( parameter_1, ..., parameter_n)  
{  
    statement_1;  
    statement_2;  
    .  
    .  
    .  
    statement_n;  
}
```

where:

The **data_type** is the type of data of the value that will be returned by the function.

The **function_name** is the identifier by which the function can be called.

The **parameters** can be as many as needed. Each parameter consists of a **data type** followed by an **identifier**, with each parameter being separated from the next by a **comma**. Each parameter looks very much like a regular variable declaration (for example: **int x**), and in fact acts within the function as a regular variable which is local to the function,

meaning it can only be used by the function. The purpose of parameters is to allow the passing of arguments to the function from the location where it is called from.


The statements consists the body of the function. It is a block of statements surrounded by **braces { }** that specify what the function actually does.

SAMPLE PROBLEM

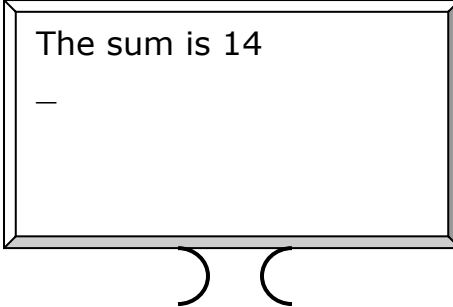
Make a C++ program that will add two numbers using function.

SAMPLE PROGRAM

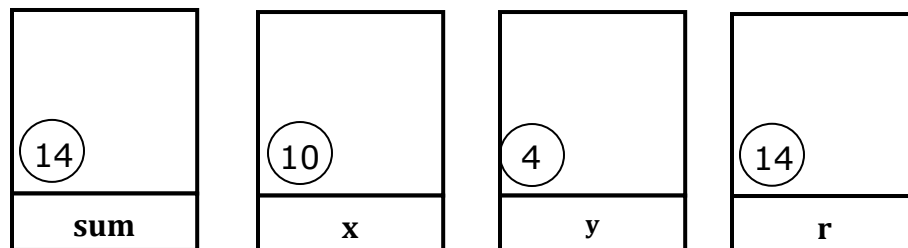
```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int addi(int x, int y)
7  {
8      int r;
9
10     r = x + y;
11
12     return r;
13 }
14
15 int main ()
16 {
17     int sum;
18
19     sum = addi(10, 4);
20
21     cout << "The sum is " << sum<<endl;
22
23     return EXIT_SUCCESS;
24 }
```



SAMPLE OUTPUT



```
The sum is 14
—
```



SIMULATION

- The main method starts in line number 15 and proceeds by declaring variable **sum** with data type **int** at **line number 17**. The compiler will allocate memory space for this variable.
- Next, variable **sum** will have the value passed by the **function** which will be called in **Line number 19**. The values **10** will be passed to variable **x** and **4** will be passed to variable **y** in function named **addi** at **Line number 6**.
- Since the function was called, it will now be processed and in **Line number 8**, variable **r** will be given a memory allocation. **Line number 10** will add the value of **x (10)** and the value of **y (4)** and store that value to **r**.
- The stored value will then be passed back to the main method and stored in variable **sum** (still at **Line number 19**).
- **Line number 21** will print the message "**The sum is**" followed by the value of **sum** and a **newline**.
- **Line number 24** ends the program.

You can also call the function a multiple number of times. The argument of the function call is not limited to literal values only. It can also be variables which hold the same data type as to the arguments of the function.

SAMPLE PROGRAM

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int subt (int a, int b)
7 {
8     int r;
9
10    r = a - b;
11
12    return r;
13 }
14
15 int main ()
16 {
17    int x=5, y=3, z, u, v, w;
18
19    z = subt (7,2);
20
21    cout<<"The first result is " <<z<<endl;
22
23    u = subt(15, 8);
24
25    cout<<"The second result is " <<u<<endl;
26
27    v = subt(x, y);
28
29    cout << "The third result is " <<v<<endl;
30
31    w = 40 + subt(x, y);
32
33    cout << "The fourth result is " <<w<< endl;
34
35    return EXIT_SUCCESS;
36 }
```

In the sample program, the statement (***z = subt (7,2);***) calls the function **subt** and passes the **value 7** to **variable a** and the **value 2** to **variable b**. The function is then executed and **2** was subtracted from **7** and the result is **5** which was then stored in **variable r**. Next, the function returns the value of **r** to the function call and stored in variable **z**.

The next line of code in the main method was executed where the following is printed on the screen:

The first result is 5

On the statement following the last executed code, function **subt** was called again and the **value of 15** was passed to **variable a** and **8** to **variable b**. The function **subt** was then executed and returned **7** which was stored in **variable u** on the main method. The next line of output is:

The second result is 7

Next statement was the call to function **subt** again, but this time, the arguments are **variables x** and **y**. Since the **value of x** is **5**, **5** was passed to **variable a** and **3** was passed to **variable b**. Then this output appeared on the screen.

The third result is 2

The last function call passed the same values as the preceding function call but this time, the returned value was added to **40** so the display was

The fourth result is 42

void FUNCTIONS

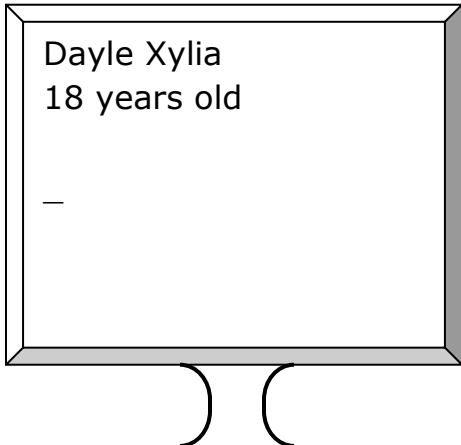
There are functions which do not return a value but only prints messages on the screen. These are called **void functions**. Void functions are created and used just like value-returning functions except they do not return a value after the function executes. In lieu of a data type, void functions use the keyword "void." A void function performs a task, and then control returns back to the caller--but it does not return a value. You may or may not use the return statement, as there is no return value.

Let's say that you want to make a function which will print your name and age on the screen.

SAMPLE PROGRAM

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 void message()
7 {
8     cout<<"Dayle Xylia"<<endl;
9     cout<<"18 years old"<<endl<<endl;
10 }
11
12 int main ()
13 {
14     message();
15
16     return EXIT SUCCESS;
17 }
```

SAMPLE OUTPUT



```
Dayle Xylia
18 years old
```

Function Overloading in C++

Function overloading is a feature of object-oriented programming (OOP) where two or more functions can have the same name but different parameters.

When a function name is overloaded with different jobs it is called Function Overloading.

In C++, two or more functions can have the same name if any of these conditions were met:

1. the number of parameters is different; and
2. the type of parameters passed is different based on its position.

These functions having the same name but different parameters are known as **overloaded functions**. For example:

```
// same name different arguments  
  
int sampleFunc()  
{  
}  
  
int sampleFunc (int a)  
{  
}  
  
float sampleFunc (double a)  
{  
}  
  
int sampleFunc (int a, double b)  
{  
}
```

All of the four functions above are overloaded functions.

Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example:

```
// Error code  
  
int sampleFunc (int a)  
{  
}  
  
float sampleFunc (int b)  
{  
}
```

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

The problem is that C++ uses the parameter list to tell the functions apart. But the parameter list of the two `sampleFunc` function is the same in count and in type based on its position in the list. The result is that C++ cannot tell these two routines apart and flags the second declaration as an error.

To understand further look at the following sample pair of functions.

```
// good code  
  
int sampleTest(float a, int b)  
{  
}  
  
float sampleTest (int b, float a)  
{  
}
```

```
// error code  
  
int sampleTest (float a, int b)  
{  
}  
  
float sampleTest (float m, int n)  
{  
}
```

Example 1: Overloading Using Different Types of Parameters

```
// Program to compute absolute value
// Works for both int and float

#include <iostream>
#include <cstdlib>

using namespace std;

// function with float type parameter
float absolute(float var)
{
    if (var < 0.0)
        var = -var;
    return var;
}

// function with int type parameter
int absolute(int var)
{
    if (var < 0)
        var = -var;
    return var;
}

int main()
{
    // call function with int type parameter
    cout << "Absolute value of -5 = " << absolute(-5) << endl;

    // call function with float type parameter
    cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;

    return EXIT_SUCCESS;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

```
float absolute(float var) { ←
    // code
}

int absolute(int var) { ←
    // code
}

int main() {
    absolute(-5); ←
    absolute(5.5f); ←
    ... ..
}
```

A diagram with red arrows indicating function calls. One arrow points from the first call in main to the float absolute function. Another arrow points from the second call in main to the int absolute function. A third arrow points from the first call in main to the int absolute function. A fourth arrow points from the second call in main to the float absolute function.

In this program, we overload the `absolute()` function. Based on the type of parameter passed during the function call, the corresponding function is called.

Example 2: Overloading Using Different Number of Parameters

```
#include <iostream>
#include <cstdlib>

using namespace std;

// function with 2 parameters
void display(int var1, double var2)
{
    cout << "Integer number = " << var1;
    cout << " and double number = " << var2 << endl;
}

// function with double type single parameter
void display(double var)
{
    cout << "Double number = " << var << endl;
}

// function with int type single parameter
void display(int var)
{
    cout << "Integer number = " << var << endl;
}

int main()
{
    int a = 5;
    double b = 5.5;

    // call function with int type parameter
    display(a);
    // call function with double type parameter
    display(b);
    // call function with 2 parameters
    display(a, b);

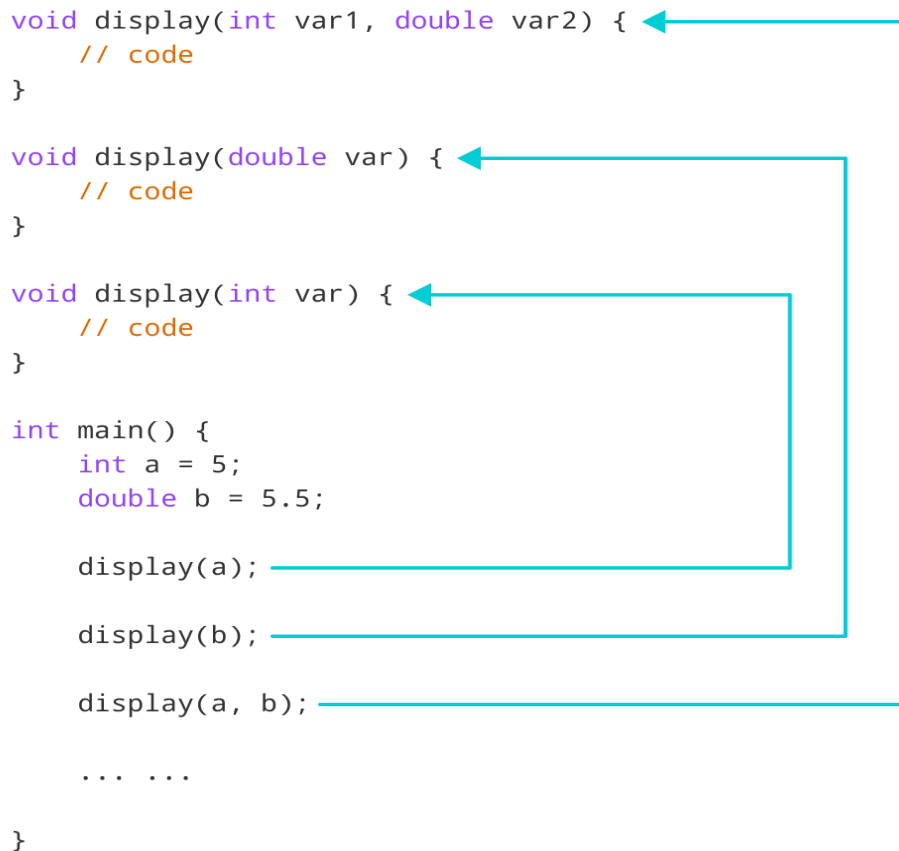
    return EXIT_SUCCESS;
}
```

Output

```
Integer number = 5  
Float number = 5.5  
Integer number = 5 and double number = 5.5
```

Here, the `display()` function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding `display()` function is called.

```
void display(int var1, double var2) {  
    // code  
}  
  
void display(double var) {  
    // code  
}  
  
void display(int var) {  
    // code  
}  
  
int main() {  
    int a = 5;  
    double b = 5.5;  
  
    display(a);  
    display(b);  
    display(a, b);  
  
    ... ..  
}
```

A diagram illustrating function overloading. It shows three function definitions for `display()` with different signatures: `display(int var1, double var2)`, `display(double var)`, and `display(int var)`. Below these is a `main()` function that calls `display(a)`, `display(b)`, and `display(a, b)`. Red arrows point from each call in `main()` to the corresponding function definition above it, showing how the compiler resolves the call based on the argument types and counts.

The return type of all these functions is the same, but that need not be the case for function overloading.

Note: In C++, many standard library functions are overloaded. For example, the `sqrt()` function can take `double`, `float`, `int`, etc. as parameters. This is possible because the `sqrt()` function is overloaded in C++.

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 10

JUMBLD WORDS. ARRANGE THE LETTERS TO FORM THE NEEDED WORDS.

1. A module or segment of code that perform an individual task.

IF NOT C NU

2. There are functions which do not return a value but only prints messages on the screen.

SOUND OF VIC TIN

3. It is a block of statements surrounded by braces { } that specify what the function actually does.

IN THE FFOOD BY COUNT

4. The purpose of these is to allow the passing of arguments to the function from the location where it is called from.

MATS ARE REP

5. This is put in the body of the program to access the function.

U FILL CANTON C

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 11

MAKE A C++ PROGRAM THAT WILL DISPLAY YOUR NAME, AGE AND BIRTHDATE ON THE SCREEN USING A *function*.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Laboratory Exercise No.7

MAKE A C++ PROGRAM THAT WILL PRINT THE AVERAGE OF THE VALUES OF THREE QUIZZES USING *function*. THE VALUES OF THE QUIZZES ARE INPUT VALUES FROM THE USER.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 12

MAKE A C++ PROGRAM THAT WILL INPUT TWO NUMBERS AND DISPLAY THE SUM OF THE NUMBERS. DO THIS FIVE TIMES. USE *function* FOR THE COMPUTATION OF THE SUM.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Laboratory Exercise No. 8

MAKE A C++ PROGRAM THAT WILL PRINT THE HIGHEST NUMBER OF THE THREE INPUT NUMBERS USING A *function*.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Seatwork No. 13

MAKE A C++ PROGRAM THAT WILL DISPLAY *NETPAY* OF AN EMPLOYEE BY INPUTTING THE *basic pay* AND *overtime pay*. IT SHOULD COMPUTE FIRST THE *gross pay* WHICH IS THE SUM OF THE INPUT VALUES AND THE *tax* WHICH IS 10% OF THE *basic pay*. THE *netpay* IS *grosspay* MINUS *tax*. USE A *function* FOR EACH COMPUTATION.

PROGRAM CODE

Seat No.: _____

Rating: _____

Name: _____

Date: _____

Laboratory Exercise No. 8

Write a C++ Program with three functions in these conditions:

1. return type is *void*.
2. function name is *display*.
3. the 1st function has two parameters with an *int type* and *float type*; when it is called it will print the phrase "Integer number = " with the inputted *int value* in the parameter. In the second line, it will print the phrase "and float number = " with the inputted *float value* in the parameter.
4. the 2nd function has one parameter with a *float type*; when it is called phrase "float number = " with the inputted *float value* in the parameter.
5. the 3rd function has one parameter with an *int type*; when it is called it will print the phrase "Integer number = " with the inputted *int value* in the parameter.

Sample output:

```
Integer number: 91
Float number: 28.7
Integer number: 91 and float number: 28.7

Process returned 0 (0x0)   execution time : 0.188 s
Press any key to continue.
```

Seat No.: _____

Rating: _____

Name: _____

Date: _____

CHAPTER TEST

I. FIND THE LISTED WORDS BELOW WHICH ARE RELATED TO FUNCTIONS AND THE SUBJECT CODE.

FUNCTIONS

DATATYPE

FUNCTION NAME

VOID

BRACES

PARAMETERS

COMMA

RETURN

FUNCTION CALL

D	R	T	G	H	Y	I	I	A	S	D	F	Y	M	N
Q	A	Z	M	I	D	U	Y	W	E	T	U	A	S	F
W	S	X	N	O	A	Y	V	I	D	G	N	R	F	X
E	F	U	N	C	T	I	O	N	S	F	C	D	K	X
R	D	C	V	L	A	T	I	J	R	E	T	U	R	N
T	F	V	X	K	T	R	D	N	A	W	I	D	C	S
Y	G	C	Z	J	Y	R	R	Z	S	S	O	U	C	D
U	H	B	O	H	P	R	B	T	S	J	N	J	1	F
P	A	R	A	M	E	T	E	R	S	G	N	W	0	G
I	J	N	A	H	M	W	R	G	A	V	A	E	2	H
O	K	M	S	G	R	A	S	D	S	C	M	E	R	J
P	V	E	E	A	R	B	S	S	S	V	E	S	T	K
F	U	N	C	T	I	O	N	C	A	L	L	S	O	L

- II. Make a C++ program that will input the first letter of a shape (*square, triangle, circle or rectangle*). It should test which area will be computed based on the input shape. There should be an *error message* if the input value is not one of the choices. Use a *function* for the computation of the area of each shape. The needed value will be passed through the function call, say for the *area of the square*, the side which will be an input from the *main function* will be passed to the *function* that will compute for the area of the square. Use *selection structure* to determine the shape of the area that will be computed.

PROGRAM CODE