Outline

**Visual Studio**: start project, debug program

**Program Structure**: iostream, std, main, sequential, return, comment

**Integer Variable**: declare, assign, initialize, arithmetic, input/output, round, unsigned, constant.

**Array Variable**: declaration, index, initialization, multi-dimensional array, sparse matrix

**Instruction**: arithmetic/increment, relational/logical, bitwise, if-else, switch, while-loop, for-loop, break

**Other Variables**: float, char, string, enum, vector, structure

**Pointer Variable**: declaration, relation to integer variable, access content, relation to integer array

**Function**: definition, declaration, assignment passing, local/global variable, overload

**Class**: definition, public/private member, constructor/destructor, declaration, memory
Instruction

### Operation
- arithmetic
- increment
- relational
- logical
- bitwise
- assignment

### Flow Control
- if-else
- switch
- for loop
- while
- do-while
- break/continue
Instruction

**Operation**
- arithmetic
- increment
- relational
- logical
- bitwise
- assignment

**Flow Control**
- if-else
- switch
- for loop
- while
- do-while
- break/continue
1. Arithmetic Operation

Perform arithmetic operations on expressions (e.g., variables).

- `exp + exp` addition
- `exp - exp` subtraction
- `exp * exp` multiplication
- `exp / exp` division
- `exp % exp` modulo (remainder)

```c
int main()
{
    int x=5, y=3, z;
    z = x + y;
    z = x - y;
    z = x * y;
    return 0;
}
```
Arithmetic Operation

EXP can be any valid expression.

- `exp + exp`  addition
- `exp - exp`  subtraction
- `exp * exp`  multiplication
- `exp / exp`  division
- `exp % exp`  modulo (remainder)

```c
int main()
{
    int x=5, y=3, z;
    z = (x+y) * (x-y);
    return 0;
}
```

These are expressions.
2. Increment and Decrement Operation

Increment increases a variable by one, and decrement decreases it by one.

- post increment = increment *after* the associated instruction is executed.
- pre increment = increment *before* the associated instruction is executed.

```
var ++    post increment
var --    post decrement
++ var    pre increment
-- var    pre decrement
```
Example 1: Post Increment vs Pre Increment

First print \( x \) on screen, then increase \( x \) by 1.

```cpp
int main()
{
    int x=5;
    cout << x++;    \quad \text{See 5 on screen. After that, } x = 6.
    cout << x;
    return 0;
}
```

Now see 6 on screen.
Example 1: Post Increment vs Pre Increment

```cpp
int main()
{
    int x = 5;
    cout << ++x;   // Pre Increment
    cout << x;    // See 6 on screen.
    return 0;
}
```

First increase `x` by 1, then print it on screen. Still see 6.
Example 2: Post Increment vs Pre Increment

```cpp
int main()
{
    int x = 5, y;
    y = ++x * 3;
    cout << y;
    return 0;
}
```

First increase x by 1, then run “y = x * 3”. Output 18 -- do you know why?
Example 2: Post Increment vs Pre Increment

Pre Increment
First run \( y = x \times 3 \),
then increase \( x \) by 1.

```cpp
int main()
{
    int x=5, y;
    y = x++ * 3;
    cout << y;
    return 0;
}
```

Output 15 -- do you know why?

Q: what is the value of \( x \) now?
3. Relational Operation

Compare two expressions. Return 1 if result is true, and return 0 if result is false.

- \( \text{exp} < \text{exp} \) less than
- \( \text{exp} > \text{exp} \) greater than
- \( \text{exp} \leq \text{exp} \) less than or equal
- \( \text{exp} \geq \text{exp} \) greater than or equal
- \( \text{exp} == \text{exp} \) equal to
- \( \text{exp} != \text{exp} \) not equal to
Example 1

```c++
int main()
{
    int x=5, y=3;
    cout << (x>y);
    cout << (x==y);
    return 0;
}
```

Output 1, because \( x>y \) is true.

Output 0, because \( x==y \) is false.
Example 2

Output 0, because
exp1: \((x+y) = 8\)
exp2: \((x-y) = 2\)
exp1 \(\leq\) exp2 is false.

Output 1, because
exp1: \((x*y) = 15\)
exp2: \((++x) = 6\)
exp1 \(!=\) exp2 is true.

```c
int main()
{
    int x=5, y=3;
    cout << ( (x+y) <= (x-y) );
    cout << ( (x*y) != (++x) );
    return 0;
}
```

We can also compare two expressions.
4. Logical Operation

Perform logical operation on expressions, which take values 0 (false) or 1 (true).

! exp  \text{ logical not}
exp \&\& exp  \text{ logical and}
exp | | exp  \text{ logical or}
Example: Logical Operator AND

```cpp
int main()
{
    int x=5, y=3;
    cout << (x>7) && (y==3); // exp1 = 0 (false)
    cout << (!(x>7) && (y==3)); // exp2 = 1 (true)
    output 0
    return 0;
}
```

Exp1 && Exp2 = 0 && 1 = 0

Exp1 && Exp2 = 0 && 1 = 0
Example: Logical Operator AND

```c++
int main()
{
    int x=5, y=3;
    cout << ( (x>7) && (y==3) );
    cout << ( !(x>7) && (y==3) );
    return 0;
}
```

- `exp1 = 0` (true)
- `exp2 = 1` (true)

Output:

- `exp1 && exp2 = 1 && 1 = 1`
- `(x>7)` is false, so its negative is true.
5. Bitwise Operation

Operate on integers at their bit level.

\[
\begin{align*}
\sim \ exp & \quad \text{bitwise complement} \\
exp \ & \& \ exp & \quad \text{bitwise and} \\
exp \ \& \ \& \ exp & \quad \text{bitwise exclusive-or} \\
exp \ | \ | \ exp & \quad \text{bitwise or} \\
\exp_1 \ \ll \ \exp_2 & \quad \text{shift } \exp_1 \ \text{left by } \exp_2 \ \text{bits} \\
\exp_1 \ \gg \ \exp_2 & \quad \text{shift } \exp_1 \ \text{right by } \exp_2 \ \text{bits}
\end{align*}
\]
An integer is expressed by 4 bytes and thus 32 bits.

$x = 1$  

32 bits, all zeros except the least significant one.
An integer is expressed by 4 bytes and thus 32 bits.

<table>
<thead>
<tr>
<th>integer</th>
<th>32-bit expression of the integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$</td>
<td>0 0 ... 0 0 1</td>
</tr>
<tr>
<td>$x = 2$</td>
<td>0 0 ... 0 1 0</td>
</tr>
<tr>
<td>$x = 3$</td>
<td>0 0 ... 0 1 1</td>
</tr>
</tbody>
</table>
Example Bitwise Operations

```c
int main()
{
    int x=1, y=2, z;
    z = x & y;
    z = x | y;
    z = x << 2;
    return 0;
}
```

- **AND Operation**
  - `z = x & y;`
  - `y`:
    |   |   |   |   |   |   |
    | 0 | 0 | ... | 0 | 1 | 0 |
  - `z`:
    |   |   |   |   |   |   |
    | 0 | 0 | ... | 0 | 0 | 0 |
  - `z = 0`
```c
int main()
{
    int x=1, y=2, z;
    z = x & y;
    z = x | y;
    z = x << 2;
    return 0;
}
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>0</th>
<th>...</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

\[ z = 3 \]
Example Bitwise Operations

```c
int main()
{
    int x=1, y=2, z;
    z = x & y;
    z = x | y;
    z = x << 2;
    return 0;
}
```

```
x  = 0 0 ... 0 0 1
z  = 0 0 ... 1 0 0
```

left shift by 2 bits.
fine in 0's

```
z = 4
```
6. Assignment Operations

```c
int main()
{
    int x = 1;
    x += 3;  // equivalent to x = x + 3;
    x *= 2;
    x -= 1;
    return 0;
}
```

x = 4 -- do you know why?
6. Assignment Operations

```c
int main()
{
    int x=1;
    x += 3;
    x *= 2;  // equivalent to x = x * 2;
    x -= 1;

    return 0;
}
```

does the same thing as `x = 8` -- do you know why?
6. Assignment Operations

```c
int main()
{
    int x = 1;
    x += 3;
    x *= 2;
    x -= 1;  // minus equal is equivalent to x = x - 1;
    return 0;
}
```

$x = 7 -- do you know why?$
Instruction

**Operation**
- arithmetic
- increment
- relational
- logical
- bitwise
- assignment

**Flow Control**
- if-else
- switch
- for loop
- while
- do-while
- break/continue
1. If-else

Examine a sequence of conditions, run instructions of the first true condition, and terminate.

- If (condition1)  
  {instructions}
  \[\text{condition in parenthesis.}\]
- else if (condition2)  
  {instructions}
  \[\text{instructions in brace.}\]
- else  
  {instructions}
  \[\text{else = all other conditions}\]
Example 1

```cpp
int main()
{
    int x = 1;
    if (x > 2) {
        cout << x;
    }
    else {
        cout << x + 1;
    }
    return 0;
}
```

Check the first condition

(x>2) is false. Check the next condition.
Example 1

```cpp
int main()
{
    int x = 1;

    if (x > 2) {
        cout << x;
    }

    else {
        cout << x + 1;
    }

    return 0;
}
```

Check the next condition

Here, "else" means (x <= 2).

"else" is true. Run "cout << x + 1;"

Terminate if-else.
Example 1

```c++
int main()
{
    int x = 1;

    if (x > 2) {
        cout << x;
    }
    else {
        cout << x+1;
    }

    return 0;
}
```
Example 2

```cpp
int main()
{
    int x=-1;
    if (x > 10) {
        cout << "x > 10";  
    }
    else if (x < 0) {
        cout << "0 > x";
    }
    else {
        cout << "0 < x < 10";
    }
    return 0;
}
```

Check the first condition.  
(x>10) is false.  
check next condition.
Example 2

```cpp
int main()
{
    int x = -1;
    if (x > 10) {
        cout << "x > 10";
    }
    else if (x < 0) {
        cout << "0 > x";
    }
    else {
        cout << "0 < x < 10";
    }
    return 0;
}
```

Check the next condition. (x<0) is true. Run “cout << ’0 > x’”; Terminate if-else.
Example 2

```c
int main()
{
    int x=-1;

    if (x > 10) {
        cout << "x > 10";
    }
    else if (x < 0) {
        cout << "0 > x";
    }
    else {
        cout << "0 < x < 10";
    }

    return 0;
}
```

If-else is done. Run “return 0”.

Here, “else” means \((x >= 0 \&\& x <= 10)\)
2. Switch

Evaluate an expression and run instructions associated to a preset value (case).

Evaluate the expression

\[
\text{switch}(\text{expression}) \{
\text{case } x:\n\quad \text{// code block}
\quad \text{break;}
\text{case } y:\n\quad \text{// code block}
\quad \text{break;}
\text{default:}
\quad \text{// code block}
\}\]

If value is x, run this block and terminate switch.

If value is y, run this block and terminate switch.

In all other cases, run this and terminate switch.

All terminations are realized by the “break” operation. We will introduce it shortly.
2. Switch

Evaluate an expression and run instructions associated to a preset value (case).

Evaluate the expression

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

- expression in ().
- Instructions in {}.
- case (value) (colon)
- instructions for case x
- break at every end
- No break for default

All terminations are realized by the “break” operation. We will introduce it shortly.
Evaluate expression, which is a variable x.

Example 1

```cpp
int main()
{
    int x = 5;

    switch (x) {
    case 1:
        cout << x + 1;
        break;

    case 5:
        cout << x + 2;
        break;

    default:
        cout << x + 3;
    }

    return 0;
}
```

Then switch is terminated. No other cases will be considered.

x = 5, run this block

output 7 -- do you know why?
Example 2

Evaluate expression, which is \(((x+5)\times2)\).

No given case is satisfied, run the default block.

```
int main()
{
    int x = 5;

    switch (((x+5)*2)) {
        case 10:
            cout << x + 1;
            break;

        case 30:
            cout << x + 2;
            break;

        default:
            cout << x + 3;
    }

    return 0;
}
```

\(\text{exp} = 20 \rightarrow \text{do you know why?}\)

\(\text{output} = 8 \rightarrow \text{do you know why?}\)
3. For-loop

Iteratively run a set of instructions until a preset condition is violated.

```
for (initialization; condition; increment) {
    instructions in every iteration;
}
```

- for () statement is used to control the loop.
- It has three parts,
  - initialization
  - condition
  - increment
- separated by semicolon.

instructions are included in braces.
Initialization

Set things up \textit{before} the loop begins. Typically initialize loop index, e.g., i = 0.

Initialization is only run for one time, no matter how many iterations follow.

\begin{verbatim}
for (initialization; condition; increment)
{
  instructions in every iteration
}
\end{verbatim}

Loop index tells the number of iteration

“i=0” means 1st round

“i=1” means 2nd round

“i=2” means 3rd round

......
Check condition *after* every iteration. Typically, bound the range of loop index, e.g., i<10.

If condition is true, run the next iteration.

```
for (initialization; condition; increment)
{
    instructions in every iteration
}
```

If condition is false, terminate the loop.
Increment

Increment at the end of every iteration, \textit{before} checking condition.

Typically increment loop index, e.g., \texttt{i++}

Incrementing index brings us to the next iteration.

\begin{verbatim}
for (initialization; condition; increment)
{
    instructions in every iteration
}
\end{verbatim}
Example: use a for-loop to print numbers 0, 1, ..., 9.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```
```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```
Yes! we can declare index inside “for ()”.

Here, we ask index to start from 0. But it can start from any number.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```
We terminate the loop when \( i < 10 \) is false, i.e., when \( i \geq 10 \).
At the end of every iteration, increment the loop index by 1.

Calculation
In total, 10 iterations.
- i starts from 0
- i++ after every round
- stop when i < 10

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```
Let’s go over the for-loop process.

```c++
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << endl;
    }
    return 0;
}
```
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";    // output 0 and change line
    }
    return 0;
}
```

Run instruction in the 1st iteration.
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {  // now i = 1
        cout << i << "\n";
    }
    return 0;
}
```

At the end of the 1st iteration, run increment i++

1st iteration is done.
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```

Check condition to decide whether we should run the next iteration.

condition $i < 10$ is true, because $i = 1$ thus run the next iteration.
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```

Run instruction in the 2nd iteration and change line output 1
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }

    return 0;
}
```

At the end of the 2nd iteration, run increment i++

2nd iteration is done.
Let’s go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << " \n";
    }
    return 0;
}
```

Check condition to decide whether we should run the next iteration.

The condition `i < 10` is true, because `i = 2` thus run the next iteration.
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";        // output 2
    }
    return 0;
}
```

Run instruction in the 3rd iteration and change line.
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << " \n";
    }
    return 0;
}
```

At the end of the 3rd iteration, run increment i++. Now i = 3

3rd iteration is done.
Let’s go over the for-loop process.

Iterate until $i = 9$…
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }

    return 0;
}
```

- **Run instruction in the 10th iteration**
- **Output 9**
- **Change line**
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {  // now i = 10
        cout << i << "\n";
    }
    return 0;
}
```

At the end of the 10th iteration, run increment i++

10th iteration is done.
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```

Check condition to decide whether we should run the next iteration.

condition \( i < 10 \) is false, since \( i = 10 \) thus terminate the loop.
Let’s go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```

For-loop is done.
Run next instruction.
Let's go over the for-loop process.

```cpp
int main()
{
    for (int i = 0; i < 10; i++) {
        cout << i << "\n";
    }
    return 0;
}
```
Scope of the loop index.

If index \( i \) is declared inside the for statement, it only exists inside the for loop.

Here, \( i \) is a local variable. We will talk more about the scope in functions.

Calling index \( i \) outside the loop will cause an error, because \( i \) does not exist outside the loop.
Nested Loop: a loop of a loop of a loop of a loop of a loop...

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }

    return 0;
}
```

Here is a loop.
Instruction of the above loop is also a loop.
For every index i, run this loop till it terminates.
Particularly useful with multi-dimensional arrays.
Each loop takes care of one dimension.
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }

    return 0;
}
```

Memory

i = 0, j does not exist.
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory:
- `i = 0, j does not exist.`
- `i = 0, j = 0`
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory

i = 0, j does not exist.
i = 0, j = 0

Run instruction and output x[0][0]
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1, 2}, {3, 4}, {5, 6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory

- i = 0, j does not exist.
- i = 0, j = 0
- i = 0, j = 1

At the end, increment j.
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }

    return 0;
}
```

Memory:
- i = 0, j does not exist.
- i = 0, j = 0
- i = 0, j = 1

Run instruction and output x[0][1]
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory:
- i = 0, j does not exist.
- i = 0, j = 0
- i = 0, j = 1
- i = 0, j = 2

At the end, increment j.
Process of a nested loop.

```cpp
text
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory:
- i = 0, j does not exist.
- i = 0, j = 0
- i = 0, j = 1
- i = 0, j = 2

Check condition.

j < 2 is false, thus terminate the inner loop.
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory:
- i = 0, j does not exist.
- i = 0, j = 0
- i = 0, j = 1
- i = 0, j = 2
- i = 1, j does not exist.

j only exists inside the inner loop.
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory

- i = 0, j does not exist.
- i = 0, j = 0
- i = 0, j = 1
- i = 0, j = 2
- i = 1, j does not exist.

Check condition.

i < 3 is true, run the next iteration.
Process of a nested loop.

```c
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory:
- i = 0, j does not exist.
- i = 0, j = 0
- i = 0, j = 1
- i = 0, j = 2
- i = 1, j does not exist.
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory:
- $i = 0$, $j$ does not exist.
- $i = 0$, $j = 0$
- $i = 0$, $j = 1$
- $i = 0$, $j = 2$
- $i = 1$, $j$ does not exist.
- $i = 1$, $j = 0$.  

Initialize $j = 0$
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Memory

- $i = 0, j$ does not exist.
- $i = 0, j = 0$
- $i = 0, j = 1$
- $i = 0, j = 2$
- $i = 1, j$ does not exist.
- $i = 1, j = 0$. 

Run instruction and output $x[1][0]$
Process of a nested loop.

Iterate over all values of index i, and for each i, over all values of index j...
Process of a nested loop.

```cpp
int main()
{
    int x[3][2] = {{1,2},{3,4},{5,6}};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << x[i][j];
        }
    }
    return 0;
}
```

Order of output

- i=0, j=0, x[0][0]=1
- i=0, j=1, x[0][1]=2
- i=1, j=0, x[1][0]=3
- i=1, j=1, x[1][1]=4
- i=2, j=0, x[2][0]=5
- i=2, j=1, x[2][1]=6
This is end of the for-loop statement.

Next we introduce two other statements: break and continue;
4. Break

"break" is used to break out of the current loop.

If run, two things are skipped:
1. the rest instructions in the current iteration
2. the rest iterations

Suppose we want to find element 3 in the array x.
We use a for-loop to go over all elements in x, but we can stop the loop once 3 is found!
This line will not be executed after break.

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }
    return 0;
}
```
Example

Initialization.

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }

        cout << x[i];
    }

    return 0;
}
```

Memory

i = 0, x[0] = 1.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }

    return 0;
}
```

Memory

i = 0, x[0] = 1.

---

Run if-else.

If condition (x[i]==3) is false, thus do not run break.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }

    return 0;
}
```

Run next instruction.

Output 1 on screen.

Memory

i = 0, x[0] = 1.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }
    return 0;
}
```

At the end of for loop, increment index i.

Memory

- i = 0, x[0] = 1.
- i = 1, x[1] = 2.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }

        cout << x[i];
    }

    return 0;
}
```

Memory

- i = 0, x[0] = 1.
- i = 1, x[1] = 2.

Check loop condition.

i < 5 is true, so run the next iteration.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1.
i = 1, x[1] = 2.

Run if-else.

If condition (x[i]==3) is false, thus do not run break.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }

        cout << x[i];
    }

    return 0;
}
```

Memory

- i = 0, x[0] = 1.
- i = 1, x[1] = 2.

Run next instruction.
Output 2 on screen.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

- \(i = 0, x[0] = 1\).
- \(i = 1, x[1] = 2\).
- \(i = 2, x[2] = 3\).
Example

Check loop condition.

i < 5 is true, so run the next iteration.

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1.
i = 1, x[1] = 2.
i = 2, x[2] = 3.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

- $i = 0$, $x[0] = 1$
- $i = 1$, $x[1] = 2$
- $i = 2$, $x[2] = 3$

Run if-else.

If condition $(x[i]==3)$ is true! thus run “break”!

The rest instructions and iterations will all be skipped.

This instruction will not be run.

Thus 3 is not output on screen.

Neither are outputs of $i=3$ & $i=4$. 
Example

```c
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            break;
        }
        cout << x[i];
    }
    return 0;
}
```

The loop is done.
Run return 0.

Only output 1 and 2.
5. Continue

Similar to break, but...

Only skip the rest instructions in the current iteration.

Do not skip the rest iterations.

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

This line will be skipped, and we continue to the next iteration.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Initialization.

Memory

i = 0, x[0] = 1.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }

        cout << x[i];
    }

    return 0;
}
```

Memory

\[ i = 0, \ x[0] = 1. \]
Example

```c++
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Run next instruction.

Output 1 on screen.

Memory

i = 0, x[0] = 1. (output)
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1. (output)
i = 1, x[1] = 2.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1. (output)
i = 1, x[1] = 2.

Check loop condition.
i < 5 is true, so run the next iteration.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Run if-else.

If condition (x[i]==3) is false, thus do not run continue.

Memory

\[
\begin{align*}
\text{i} & = 0, \ x[0] = 1. \ (\text{output}) \\
\text{i} & = 1, \ x[1] = 2.
\end{align*}
\]
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1. (output)
i = 1, x[1] = 2. (output)
Example

```
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1. (output)
i = 1, x[1] = 2. (output)
i = 2, x[2] = 3.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1. (output)
i = 1, x[1] = 2. (output)
i = 2, x[2] = 3.
Example

Run if-else.

If condition \((x[i]==3)\) is true!
thus run continue!

Skip the rest instructions!

```
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

- \(i = 0, x[0] = 1.\) (output)
- \(i = 1, x[1] = 2.\) (output)
- \(i = 2, x[2] = 3.\)

This will be skipped.
Thus 3 is not output on screen.
Continue to the next iteration.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

- $i = 0$, $x[0] = 1$. (output)
- $i = 1$, $x[1] = 2$. (output)

Increment index $i$. 

Example

Check loop condition.

i < 5 is true, so run the next iteration.

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

i = 0, x[0] = 1. (output)
i = 1, x[1] = 2. (output)
i = 2, x[2] = 3.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

- i = 0, x[0] = 1. (output)
- i = 1, x[1] = 2. (output)
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }
    return 0;
}
```

Memory

- $i = 0, x[0] = 1. \text{ (output)}$
- $i = 1, x[1] = 2. \text{ (output)}$
- $i = 3, x[3] = 4. \text{ (output)}$

Run next instruction.

Output 4 on screen.
Example

```cpp
int main()
{
    int x[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (x[i] == 3) {
            continue;
        }
        cout << x[i];
    }

    return 0;
}
```
Output all but 3.
This is end of the break and continue statements.

Next we introduce two other statements: break and continue;
6. while

Iteratively run a set of instructions while the condition is true.

While (condition) ←-- condition in parenthesis.

{instructions} ←-- instructions in brace.
Example

```cpp
int main()
{
    int i = 1;

    while (i < 5) {
        cout << i << ' \n';
        i = i + 2;
    }

    return 0;
}
```

Iterate over these instructions while condition “i < 5” is true.
Example

```
int main()
{
    int i = 1;

    while (i < 5) {
        cout << i << 'n';
        i = i + 2;
    }

    return 0;
}
```

Initialize i = 1.
 Example

 int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << 'n';
        i = i + 2;
    }
    return 0;
}
```cpp
int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << ' \n';
        i = i + 2;
    }
    return 0;
}
```
Example

```cpp
int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << ' \n';
        i = i + 2;
    }
    return 0;
}
```

Memory

i = 1 (output)
i = 3
Example

```cpp
int main()
{
    int i = 1;

    while (i < 5) {
        cout << i << 'n';
        i = i + 2;
    }

    return 0;
}
```

Condition (i<5) is true, thus run one iteration.

Check condition.

Memory

i = 1 (output)
i = 3
```cpp
int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << ' \n';
        i = i + 2;
    }
    return 0;
}
```
```cpp
int main()
{
    int i = 1;

    while (i < 5) {
        cout << i << 'n';
        i = i + 2;
    }

    return 0;
}
```

Example

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1 (output)</td>
</tr>
<tr>
<td>i = 3 (output)</td>
</tr>
<tr>
<td>i = 5</td>
</tr>
</tbody>
</table>

Run \(i = i + 2\).
Example

Check condition.

Condition \((i<5)\) is false!
Thus terminate while loop.

```cpp
int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << 'n';
        i = i + 2;
    }
    return 0;
}
```

Memory

- \(i = 1\) (output)
- \(i = 3\) (output)
- \(i = 5\)
Example

```cpp
int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << 'n';
        i = i + 2;
    }
    return 0;
}
```

Memory

- i = 1 (output)
- i = 3 (output)
- i = 5
Need instruction to update condition in while-loop.

```cpp
int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << ' \n';
        i = i + 2;
    }
    return 0;
}
```

In while-loop, it is important to include an instruction that can update the condition.

At the end of every iteration, increment i by 2. (i is evaluated at condition)
Need instruction to update condition in while-loop.

```cpp
int main()
{
    int i = 1;

    while (i < 5) {
        cout << i << '\n';
        i = i + 2;
    }

    return 0;
}
```

Q: what if we remove this line?
7. do-while

Similar to while-loop, but first run instruction and then check condition.

First run instructions. \{instructions\}

Then check condition. \textbf{While (condition);} \quad \text{Semicolon after condition.}
Example

```
int main()
{
    int i = 1;
    do {
        cout << i << " \n";
        i = i + 2;
    } while (i < 5);
    return 0;
}
```
Example

```
int main()
{
    int i = 1;
    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```
Example

```cpp
int main()
{
    int i = 1;
    do {
        cout << i << '
';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```

Run cout.

No need to check any condition before running the 1st iteration.

Memory

i = 1 (output)
Example

```
int main()
{
    int i = 1;
    do {
        cout << i << 'n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```

Memory

i = 1 (output)
i = 3
Example

```cpp
int main()
{
    int i = 1;
    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```

Check condition

(i<5) is true, so run the next iteration.

Memory

i = 1 (output)

i = 3
Example

```cpp
int main()
{
    int i = 1;
    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```

Memory

i = 1 (output)
i = 3 (output)
Example

```cpp
int main()
{
    int i = 1;

    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);

    return 0;
}
```

Memory

- $i = 1$ (output)
- $i = 3$ (output)
- $i = 5$
Example

```
int main()
{
    int i = 1;
    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```

Check condition

(i<5) is false!
Terminate the loop.

Memory

- i = 1 (output)
- i = 3 (output)
- i = 5
Example

```cpp
int main()
{
    int i = 1;
    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```

Memory

- i = 1 (output)
- i = 3 (output)
- i = 5
Comparison between “while” and “do-while”

Q: is there a difference between the following two outputs?

```cpp
int main()
{
    int i = 1;
    while (i < 5) {
        cout << i << '\n';
        i = i + 2;
    }
    return 0;
}
```

```cpp
int main()
{
    int i = 1;
    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```
Comparison between “while” and “do-while”

Q: what about now?

```c++
int main()
{
    int i = 5;
    while (i < 5) {
        cout << i << '\n';
        i = i + 2;
    }
    return 0;
}
```

```c++
int main()
{
    int i = 5;
    do {
        cout << i << '\n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```
Comparison between “while” and “do-while”

Q: what about now?

```cpp
int main()
{
    int i = 5;
    while (i < 5)
    {
        cout << i << 'n';
        i = i + 2;
    }
    return 0;
}
```

False; stop. Output 5.

```cpp
int main()
{
    int i = 5;
    do {
        cout << i << 'n';
        i = i + 2;
    } while (i < 5);
    return 0;
}
```

First, run

False; stop. No output.
Final subject: operator precedence -- who goes first?

**Operation**
- arithmetic
- increment
- relational
- logical
- bitwise
- assignment

**Flow Control**
- if-else
- switch
- for loop
- while
- do-while
- break/continue
Operator Precedence

Operators on the top have higher priority than those at the bottom.

Parenthesis can change priority.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++   var--</td>
</tr>
<tr>
<td>dereference/address</td>
<td>++var   --var   +exp   -exp   ~exp   !exp</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>*       /       %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+       -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt;      &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt;       &lt;=     &gt;     &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>==      !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>=     +=     -=     *=     /=     %=     &gt;&gt;=     &lt;&lt;=     &amp;=     ^=</td>
</tr>
</tbody>
</table>
int main()
{
    int i = 1;
    cout << (i++ * 5 > 7);
    return 0;
}
Identify all operators.

```c++
int main()
{
    int i = 1;
    cout << ( i++ * 5 > 7 );
    return 0;
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting function call</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++ var--</td>
</tr>
<tr>
<td>prefix operators</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>dereference/address</td>
<td>*pointer &amp;var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
Example 1

```c++
int main()
{
    int i = 1;
    cout << (i++ * 5 > 7);
    return 0;
}
```

Note i will not increment immediately; wait for the associated operation which has the next highest priority.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++  var--</td>
</tr>
</tbody>
</table>
| prefix operators/dereference/address | ++var --var +exp -exp ~exp !exp *
| multiplication/division      | *  /  %                                       |
| addition/subtraction         | +  -                                          |
| shift                         | <<  >>                                        |
| comparison                   | <  <=  >  >=                                  |
| equality                     | ==  !=                                        |
| bitwise and                   | &                                             |
| bitwise exclusive-or         | ~                                             |
| bitwise or                    | |                                             |
| logical and                   | &&                                            |
| logical or                    | ||                                             |
| conditional                  | bool_exp ? true_exp : false_exp                |
| assignment                   | +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=     |
Example 1

```cpp
int main()
{
    int i = 1;
    cout << ( i++ * 5 > 7 );
    return 0;
}
```

After this $i = 2$.  

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++ var--</td>
</tr>
<tr>
<td>prefix operators</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>dereference/address</td>
<td>*pointer &amp;var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
### Example 1

```cpp
int main()
{
    int i = 1;
    cout << (i++ * 5 > 7);
    return 0;
}
```

Output 0.

This is result of $1 \times 5$. 

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[expr]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td></td>
<td>*pointer &amp;var</td>
</tr>
<tr>
<td>prefix operators</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>dereference/address</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp; &amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>+= -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
Example 2

```c++
int main()
{
    int x = 1, y = 0;
    cout << ( x++ * 3 > ++y << 2 );
    return 0;
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace.name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class.name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++, var--</td>
</tr>
<tr>
<td>prefix operators/dereference/address</td>
<td>++var, --var, +exp, -exp, ~exp, !exp</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>*, /, %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+, -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>==, !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>=, +=, -=, *=, /=, %=, &gt;&gt;=, &lt;&lt;=, &amp;=, ^=,</td>
</tr>
</tbody>
</table>
Identify all operators.

```cpp
int main()
{
    int x = 1, y = 0;
    cout << (x++ * 3 > ++y << 2);
    return 0;
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>++var, --var, +exp, -exp, ~exp, !exp</td>
</tr>
<tr>
<td>dereference/address</td>
<td>*pointer, &amp;var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>*, /, %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+, -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt;=, &gt;&gt;=</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>==, !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>=, +=, -=, *=, /=, %=, &gt;&gt;&gt;, &lt;&lt;=, &amp;=, ^=,</td>
</tr>
</tbody>
</table>
Example 2

```c
int main()
{
    int x = 1, y = 0;
    cout << (x++ * 3) + y << 2);
    return 0;
}
```

This does not mean \( x = 2 \) right away, because it is post increment.

Wait for associated operation to run.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++ var--</td>
</tr>
<tr>
<td>prefix operators</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>dereference/address</td>
<td>*pointer &amp;var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>+= -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
Example 2

```cpp
int main()
{
    int x = 1, y = 0;
    cout << (x++ * 3 > ++y) << 2);
    return 0;
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++ var--</td>
</tr>
<tr>
<td>prefix operators/dereference/address</td>
<td>++var --var +exp -exp ~exp !exp *pointer var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>~</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>

y = 1
Example 2

```c
int main()
{
    int x = 1, y = 0;
    cout << (x++ * 3 > ++y << 2);
    return 0;
}
```

After this, `x = 2`
```cpp
int main()
{
    int x = 1, y = 0;
    cout << (x++ * 3 > ++y << 2);
    return 0;
}
```

Left expression is 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++ var--</td>
</tr>
<tr>
<td>prefix operators/dereference/address</td>
<td>++var --var +exp -exp ^exp !exp *pointer &amp;var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
Example 2

```cpp
int main()
{
    int x = 1, y = 0;
    cout << ( x++ * 3 > ++y << 2 );
    return 0;
}
```

Result is false. Output 0.

Result of 1*3.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name::member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++, var--</td>
</tr>
<tr>
<td>prefix operators</td>
<td>++var, --var, +exp, -exp, ~exp, !exp</td>
</tr>
<tr>
<td>dereference/address</td>
<td>*pointer, &amp;var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>*, /, %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+, -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>==, !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>=, +=, -=, *=, /=, %=, &gt;&gt;=, &lt;&lt;=, &amp;=, ^=, ^=</td>
</tr>
</tbody>
</table>
We can change precedence using parenthesis.

Case 1: with no parenthesis.

```cpp
tint main()
{
    int x = 3, y = 1;
    cout << (x++ * 3 + y << 2);
    return 0;
}
```

We can change precedence using parenthesis.

Case 1: with no parenthesis.

```c++
int main()
{
    int x = 3, y = 1;
    cout << (x++ * 3 + y << 2);
    return 0;
}
```
We can change precedence using parenthesis.

Case 1: with no parenthesis.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << (x++ * 3 + y << 2);
    return 0;
}
```

1 * 3 = 3
We can change precedence using parenthesis.

Case 1: with no parenthesis.

```c++
int main()
{
    int x = 3, y = 1;
    cout << (x++ * 3 + y) << 2);
    return 0;
}
```

3 + 1 = 4
We can change precedence using parenthesis.

Case 1: with no parenthesis.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << (x++ * 3 + y << 2);
    return 0;
}
```

Shift 4 by 2 bits. result is 16.
We can change precedence using parenthesis.

Case 2: with parentheses.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << ( x++ * ( 3 + (y << 2) ) );
    return 0;
}
```

Rule 1: run inner parenthesis first.
Rule 2: in a parenthesis, follow the table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>var++ var--</td>
</tr>
<tr>
<td>prefix operators</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>dereference/address</td>
<td>*pointer &amp;var</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>~</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>+= -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
We can change precedence using parenthesis.

Case 2: with parentheses.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << ( x++ * ( 3 + (y << 2) ) );
    return 0;
}
```

First priority; second priority; third priority.
We can change precedence using parenthesis.

Case 2: with parentheses.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << ( x++ * ( 3 + (y << 2) ) );
    return 0;
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope resolution</td>
<td>namespace_name :: member</td>
</tr>
<tr>
<td>selection/subscripting</td>
<td>class_name.member pointer-&gt;member array[exp]</td>
</tr>
<tr>
<td>function call</td>
<td>function(args)</td>
</tr>
<tr>
<td>postfix operators</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>prefix operators</td>
<td>*pointer &amp;var</td>
</tr>
<tr>
<td>dereference/address</td>
<td>++var --var +exp -exp ~exp !exp</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>* / %</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive-or</td>
<td>^</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>bool_exp ? true_exp : false_exp</td>
</tr>
<tr>
<td>assignment</td>
<td>+= -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
We can change precedence using parenthesis.

Case 2: with parentheses.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << ( x++ * ( 3 + (y << 2) ) );
    return 0;
}
```

3 + 4 = 7.
We can change precedence using parenthesis.

Case 2: with parentheses.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << ( x++ * ( 3 + (y << 2) ) );
    return 0;
}
```
We can change precedence using parenthesis.

Case 2: with parentheses.

```cpp
int main()
{
    int x = 3, y = 1;
    cout << ( x++ * ( 3 + (y << 2) ) );
    return 0;
}
```

3 * 7 = 21. (output)

After this, x = 4.