

C++

Operators & Syntax

*originally developed by Weiguang Cui,
refined by Alexander Knebe*



Arithmetic



Comparison



Logical /
Bitwise



Assignment



Conditions



Loops

Arithmetic Operators

Op	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$x++$ / $++x$
--	Decrement	Decreases the value of a variable by 1	$x--$ / $--x$

Practice — Arithmetic (int & double)



```
int a = 2;  int b = 3;  int c;  
double d = 0.25;  double e = 1.55;  double f;
```

1

```
f = a+b;  c = d+e;
```

What are the values of f and c?

2

```
c = a/b;  f = a/b;
```

What are the values of c and f? Is there a difference?

3

```
c = e%d;  f = e%d;
```

What are the values? (Hint: think about types)

4

```
c = a++;  f = d++;
```

What are the values? Does the order of ++ matter?

5

```
c = ++a;  f = ++d;
```

Any difference compared to item 4?

6

```
cout<<a++<<endl;  cout<<++b<<endl;
```

What's different between post- and pre-increment here?

play with ++/-- in all possible ways!

Practice — Arithmetic (bool)



```
bool bot = true;  bool bof = false;
```

1

What does `cout << bot << bof` print? (Hint: try it!)

2

Can you initialize `bot` or `bof` with integers? Try 0, 1, and 3.

3

What are the results of `bot+bof`, `bot-bof`, `bot*bof`, `bot/bof`? Try each.

Practice — Arithmetic (char & string)

char

```
char a = 'c'; char b = 'd';
```

- Can you do `cout << a+b?` `cout << a-b?`
- What if `a='2'` and `b='4'`? Same operations?

string

```
string a = "c"; string b = "d";
```

- Try all the same operations. What works, what doesn't?
- What are the results compared to char?

Converting string/char to int:

```
atoi(charVar)    // C-style           stoi(stringVar)    // Modern C++ (preferred)
```

Comparison Operators

Op	Name	Returns true when...	Example
<code>==</code>	Equal to	Both values are identical	<code>x == y</code>
<code>!=</code>	Not equal	Values are different	<code>x != y</code>
<code>></code>	Greater than	Left is larger than right	<code>x > y</code>
<code><</code>	Less than	Left is smaller than right	<code>x < y</code>
<code>>=</code>	Greater or equal	Left is larger than or equal	<code>x >= y</code>
<code><=</code>	Less or equal	Left is smaller than or equal	<code>x <= y</code>

Comparison operators always return a bool value: true (1) or false (0).

Practice — Comparison Operators

Numbers

```
int a = 5, b = 3;  
cout << (a > b);    // what prints?
```

Comparing numbers is straightforward — try all six operators and observe the output.

char

```
char c1 = 'a', c2 = 'b';  
cout << (c1 == c2);
```

Can you compare char variables with == and >? What does the ordering mean for characters?

string

```
string s1 = "it is a dog";  
string s2 = "itisadog";  
cout << (s1 == s2);
```

Are "it is a dog" and "itisadog" equal? Try > and < too. What is string comparison based on?

Logical Operators

Logical Operators

Op	Name	Description	Example
&&	Logical AND	true if BOTH statements are true	x<5 && x<10
	Logical OR	true if AT LEAST ONE statement is true	x<5 x<4
!	Logical NOT	Reverses the result — true becomes false	!(x<5 && x<10)

Bitwise Operators

Bitwise Operators

Op	Name	Description	Example
&	Bitwise AND	AND each bit pair	11 & 10 → 10
	Bitwise OR	OR each bit pair	x y
^	Bitwise XOR	1 if bits differ; 0^0 and 1^1 both = 0	x ^ y
~	Bitwise NOT	Flips all bits (0→1, 1→0)	~x
<<	Left Shift	Shift bits left by N (fill with 0s)	x << N
>>	Right Shift	Shift bits right by N (discard leftmost)	x >> N

Assignment Operators

Op	Example	Equivalent
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 3</code>	<code>x = x + 3</code>
-=	<code>x -= 3</code>	<code>x = x - 3</code>
*=	<code>x *= 3</code>	<code>x = x * 3</code>
/=	<code>x /= 3</code>	<code>x = x / 3</code>
%=	<code>x %= 3</code>	<code>x = x % 3</code>

Op	Example	Equivalent
&=	<code>x &= 3</code>	<code>x = x & 3</code>
 =	<code>x = 3</code>	<code>x = x 3</code>
^=	<code>x ^= 3</code>	<code>x = x ^ 3</code>
>>=	<code>x >>= 3</code>	<code>x = x >> 3</code>
<<=	<code>x <<= 3</code>	<code>x = x << 3</code>

What about `~=` ? And can you use comparison (`==`, `!=`) or logical (`&&`, `||`) operators on the right side?

Conditions — if / else if / else



```
if (condition) {  
    // runs if condition is true  
}  
  
if (condition) {  
    // runs if condition is true  
} else {  
    // runs if condition is false  
}  
  
if (condition1) {  
    // condition1 is true  
} else if (condition2) {  
    // condition1 false, condition2 true  
} else {  
    // both false  
}
```

Key Points

if

Runs the block only when the condition evaluates to true.

else

Fallback block — runs when the if condition is false.

else if

Chain as many as you need. Only the first true branch runs.

Ternary

`var = (condition) ? ifTrue : ifFalse;`
A compact one-liner for simple choices.

How many else if can you chain? Try it!

Practice — if / else

Write a program that reads a character and identifies its type.

```
● ● ●
#include <iostream>
#include <cctype>    // isdigit, isalpha, ...
using namespace std;

bool isDigit(char ch) {
    return ch >= '0' && ch <= '9';
}

int main() {
    char ch;
    cin >> ch;

    if (isDigit(ch))      cout << "digit";
    else if (isalpha(ch)) cout << "letter";
    else                  cout << "special";

    return 0;
}
```

Your Tasks

- 1 Read a char with cin.
- 2 Print whether it is a digit, letter, or special character.
- 3 Try isdigit() from <cctype> AND the manual function above — same result?
- 4 Extend: distinguish between + * (. and other specials.

Conditions — switch



```
switch (expression) {  
    case x:  
        // code block  
        break;    // what happens if you remove this?  
    case y:  
        // code block  
        break;  
    default:    // what if you remove this?  
        // code block  
}
```

Key Questions

- 1 What happens when you remove break from a case?
- 2 What does default handle?
- 3 Can you remove default? When would that be safe?
- 4 Replace your previous if/else exercise with switch!

Loops — while / do...while



```
// while: checks condition FIRST
while (condition) {
    // runs as long as condition is true
    // WARNING: infinite loop if condition never becomes
    false!
}

// do...while: runs body FIRST, then checks
do {
    // always executes at least once
} while (condition); // same warning applies!
```

Difference

while

Condition checked BEFORE each iteration. Body may never run.

do...while

Body runs FIRST. Condition checked after. Body runs at least once.

Danger!

If the condition never becomes false → infinite loop.
Always make sure something inside changes the condition.

Practice: Use while or do...while to calculate the sum of $1 + 2 + \dots + 1000$

Loops — for / range-for



```
// Classic for loop
for (init; condition; update) {
    // code block
}
// init      → runs once before the loop
// condition → checked before each iteration
// update    → runs after each iteration

// Range-based for (C++11+) – loop over arrays
for (type varName : arrayName) {
    // code block – varName is each element
}
```

The Three Parts

Statement 1

Executed once. Typically declares the loop variable, e.g. `int i = 0`.

Statement 2

Checked before each iteration. Loop stops when false.

Statement 3

Executed after each iteration. Typically increments the variable.

Range-for

Cleaner syntax when looping over all elements of an array or container.

Practice: Sum 1–1000 with for.

Then try: `int v10[10]={0,100,200,...,900}`; loop over it with range-for, and nest another for loop inside.

break and continue

```
for (int i = 0; i < 10; i++) {  
    if (i == 4) {  
        continue; // skip this iteration  
        // swap for break → exits the whole loop!  
    }  
    cout << i << "\n";  
}
```

Try it!

Run the code with continue.
What is the output?

Now swap continue for break.
What changes?

What if the if condition was `i%2==0`?

break

Immediately exits the entire loop. Execution continues after the closing }.

```
Output stops at 3.
```

continue

Skips the rest of the current iteration and jumps to the next one.

```
Skips 4; prints 0,1,2,3,5,6,7,8,9.
```