C++_Lecture Note 3

Constant Expressions, Statements, and Operators

Constants

A constant, like a variable, is a memory location where a value can be stored. Unlike variables, constants never change in value. You must initialize a constant when it is created. C++ has two types of constants: **literal and symbolic**. A literal constant is a value typed directly into your program wherever it is needed. For example, consider the following statement: long width = 5; This statement assigns the integer variable width the value 5. The 5 in the statement is a literal constant. You can't assign a value to 5, and its value can't be changed. The values true and false, which are stored in bool variables, also are literal constants.

A symbolic constant is a constant represented by a name, just like a variable. The const keyword precedes the type, name, and initialization. Here's a statement that sets the point reward for killing a zombie:
const int KILL_BONUS = 5000;

Defining Constants
There's another way to define constants that dates back to early versions of the C language, the precursor of C++. The preprocessor directive #define can create a constant by specifying its name and value, separated by spaces: #define KILLBONUS 5000 The constant does not have a type such as int or char. The #define directive enables a simple text substitution that replaces every instance of KILLBONUS in the code with 5000. The compiler sees only the end result. Because these constants lack a type, the compiler cannot ensure that the constant has a proper value.

Enumerated Constants
Enumerated constants create a set of constants with a single statement. They are defined with the keyword enum followed by a series of comma-separated names surrounded by braces:
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
This statement creates a set of enumerated constants named COLOR with five values named RED, BLUE, GREEN, WHITE and BLACK. The values of enumerated constants begin with 0 for the first in the set and count upwards by 1. So RED equals 0, BLUE equals 1, GREEN equals 2, WHITE equals 3, and BLACK equals 4. All the values are integers.

Constants also can specify their value using an = assignment operator: enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 }; This statement sets RED to 100, GREEN to 500, and BLACK to 700. The members of the set without assigned values will be 1 higher than the previous member, so BLUE equals 101 and WHITE equals 501. The advantage of this technique is that you get to use a symbolic name such as BLACK or WHITE rather than a possibly meaningless number such as 1 or 700.

Statements

All C++ programs are made up of statements, which are commands that end with a semicolon. Each statement takes up one line by convention, but this is not a requirement, multiple statements could be put on a line as long as each ends with a semicolon. A statement controls the program's sequence of execution, evaluates an expression, or can even do nothing (the null statement).

A common statement is an assignment: x = a + b;

This statement assigns the variable x to equal the sum of a + b. The assignment operator = assigns the value on the right side of the operator to a variable on the left side. If a equals 4 and b equals 13, x will equal 17 after the statement is executed.

Whitespace

In the source code of a C++ program, any spaces, tabs, and newline characters are called whitespace. The compiler generally ignores whitespace, which serves the purpose of making the code more readable to programmers.

The assignment statement could be written in the following two ways and still work the same way:

```
x=a+b;
x      =      a      +      b      ;
```

The compiler ignores whitespace (or the lack of it). Whitespace cannot be used inside a variable name, so the variable **playerScore** could not be referred to as **player Score**. The tabs or spaces that serve the purpose of indentation in programs are whitespace. Proper indentation makes it easier to see when a program block or function block begins and ends.

Compound Statements

Several statements can be grouped together as a compound statement, which begins with an opening brace { and ends with a closing brace }. A compound statement can appear anywhere a single statement could. Although every statement in a compound statement must end with a semicolon, the compound statement itself does not end with a semicolon. Here's an example:

```
{
    temp = a;
    a = b;
    b = temp;
}
```

This compound statement swaps the values in the variables a and b using a variable named temp as a temporary holding place for one value.

Expressions

An expression is any part of a statement that returns a value, as in this simple example:
x = y + 13;
This statement makes the variable x equal to the variable y plus 13. So, if y equals 20, x equals 33. The entire statement also returns the final value of x, so it's also an expression.
To understand this better, consider a more complex statement: z = x = y + 13;
This statement consists of three expressions: .

- The expression y + 13 is stored in the variable x.
- The expression x = y + 13 returns the value of x, which is stored in the variable z.
- The expression z = x = y + 13 returns the value of z, which is not stored.

The assignment operator = causes the operand on the left side of the operator to have its value changed to the value on the right side of the operator. Operand is a mathematical term referring to the part of an expression operated upon by an operator.
The Expression program below displays the values of three variables before and after they are used in a complex multiple-expression statement.

```
 1: #include <iostream>
 2: int main()
 3: {
 4:     int x = 0, y = 72, z = 0;
 5:     std::cout << "Before\n\nx: " << x << " y: " << y;
 6:     std::cout << " z: " << z << "\n\n";
 7:     z = x = y + 13;
 8:     std::cout << "After\n\nx: " << x << " y: " << y;
 9:     std::cout << " z: " << z << "\n";
10:     return 0;
11: }
```

The above
3 program produces the following output:

```
Before x: 0 y: 72 z: 0
After x: 85 y: 72 z: 85
```

Three variables are declared and given initial values, which are displayed on lines 5–6. In line 7, expressions assign values to x and z, in that order. The new values are displayed in lines 8–9.

Operators

An operator is a symbol that causes the compiler to take an action such as assigning a value or performing multiplication, division, or another mathematical operation.

Assignment Operator

An expression consists of an assignment operator, an operand to its left called an l-value, and an operand to its right called an r-value. In the expression grade = 95, the l-value is grade, and the r-value is 95. Constants are r-values but cannot be l-values. The expression 95 = grade is not permitted in C++ because the constant 95 cannot be assigned a new value. The primary reason to learn the terms l-value and r-value is because they may appear in compiler error messages.

Mathematical Operators
There are five mathematical operators: addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). C++, like C, does not have an exponentiation operator to raise a value to a specified power.
There is a function to perform the task. Addition, subtraction, and multiplication act as you'd expect, but division is more complex.

Integer division differs from ordinary division. When you divide 21 by 4, the result is a real number that has a fraction or decimal value. By contrast, integer division produces only integers, so the remainder is dropped. The value returned by 21 / 4 is 5.

The modulus operator % returns the remainder value of integer division, so 21 % 4 equals 1. The integer division 21 / 4 is 5, leaving a remainder of 1.

When describing an expression using the modulus operator, it is called modulo, so 21 % 4 is "21 modulo 4." Modulo is the operation performed by the modulus operator and the result is called the modulus.

Finding the modulus can be useful in programming. If you want to display a statement every 10th time that a task is performed, the expression taskCount % 10 can watch for this. The modulus ranges in value from 0 to 10. Every time it equals 0, the count of tasks is a multiple of 10.

Floating-point division is comparable to ordinary division. The expression 21 / 4.0 equals 5.25.

C++ decides which division to perform based on the type of the operands. If at least one operand is a floating-point variable or literal, the division is floating point. Otherwise, it is integer division.

Combining Operators
It is not uncommon to want to add a value to a variable and then to assign the result back into the variable.
The following expression adds 10 to the value of a variable named score: score = score + 10;
This expression takes the existing value of score, adds 10 to it, and stores the result in score.
This can be written more simply using the += self-assigned addition operator: score += 10;

The self-assigned addition operator += adds the r-value to the l-value, and then assigns the result to the l-value. There are self-assigned subtraction (-=), division (/=), multiplication (*=), and modulus (%=) operators, as well.
These self-assignment operators do the same thing as longer expressions, so either form can be used at your discretion.

Increment and Decrement Operators
The most common value to add or subtract from a variable is 1. Increasing a variable by 1 is called incrementing, and decreasing it by 1 is called decrementing.
C++ includes a ++ increment operator and — decrement operator to accomplish these tasks:
score++;
zombies—;
These statements increase score by 1 and decrease zombies by 1, respectively. They are equivalent to these more verbose statements: score = score + 1; zombies = zombies – 1;

Prefix and Postfix Operators
The increment operator ++ and decrement operator — can be used either before or after a variable's name to achieve different results.

An operator placed before a variable's name is called a prefix operator, as in this statement: ++count; An operator placed after the variable name is called the postfix operator: count++;

In simple statements like the preceding examples, the operators accomplish the same thing. The count variable is increased by 1 in both statements.

The reason for the existence of prefix and postfix operators become apparent in complex expressions where a variable is being incremented or decremented and assigned to another variable. The prefix operator occurs before the variable's value is used in the expression. The postfix is evaluated after.

This will make more sense with a concrete example:
int x = 5;
int sum = ++x;

After these statements are executed, the x variable and sum variable both equal 6. The prefix operator in ++x causes x to be incremented from 5 to 6 before it is assigned to sum.
Compare it to this example:
int x = 5;
int sum = x++;
This causes sum to equal 5 and x to equal 6.

The postfix operator causes x to be assigned to sum before it is incremented from 5 to 6.

The program below contains the Years program, which counts forward several years using prefix and postfix increment operators.

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     int year = 2010;
 6:     std::cout << "The year " << ++year << " passes.\n";
 7:     std::cout << "The year " << ++year << " passes.\n";
 8:     std::cout << "The year " << ++year << " passes.\n";
 9:
10:     std::cout << "\nIt is now " << year << ".";
11:     std::cout << " Have the Seattle Mariners won the World Series yet?\n";
12:
13:     std::cout << "\nThe year " << year++ << " passes.\n";
14:     std::cout << "The year " << year++ << " passes.\n";
15:     std::cout << "The year " << year++ << " passes.\n";
16:
17:     std::cout << "\nSurely the Mariners have won the Series by now.\n";
18:     return 0;
19: }
```

This program displays the following output:

```
The year 2011 passes.
The year 2012 passes.
The year 2013 passes.

It is now 2013. Have the Seattle Mariners won the World Series yet?

The year 2013 passes.
The year 2014 passes.
The year 2015 passes.
```

The Years program counts forward the years, anticipating the first World Series victory by the Seattle Mariners, one of only two Major League Baseball franchises to never reach the World Series. The program begins by setting the year variable to 2010 in Line 5.

Line 6 produces the first output of the program: “The year 2011 passes.” Take note that the year is 2011, not 2010 as it was originally set. This happens because the prefix operator in that line changes the value of year before it is displayed.

Several years pass, and in line 10, the year equals 2013.

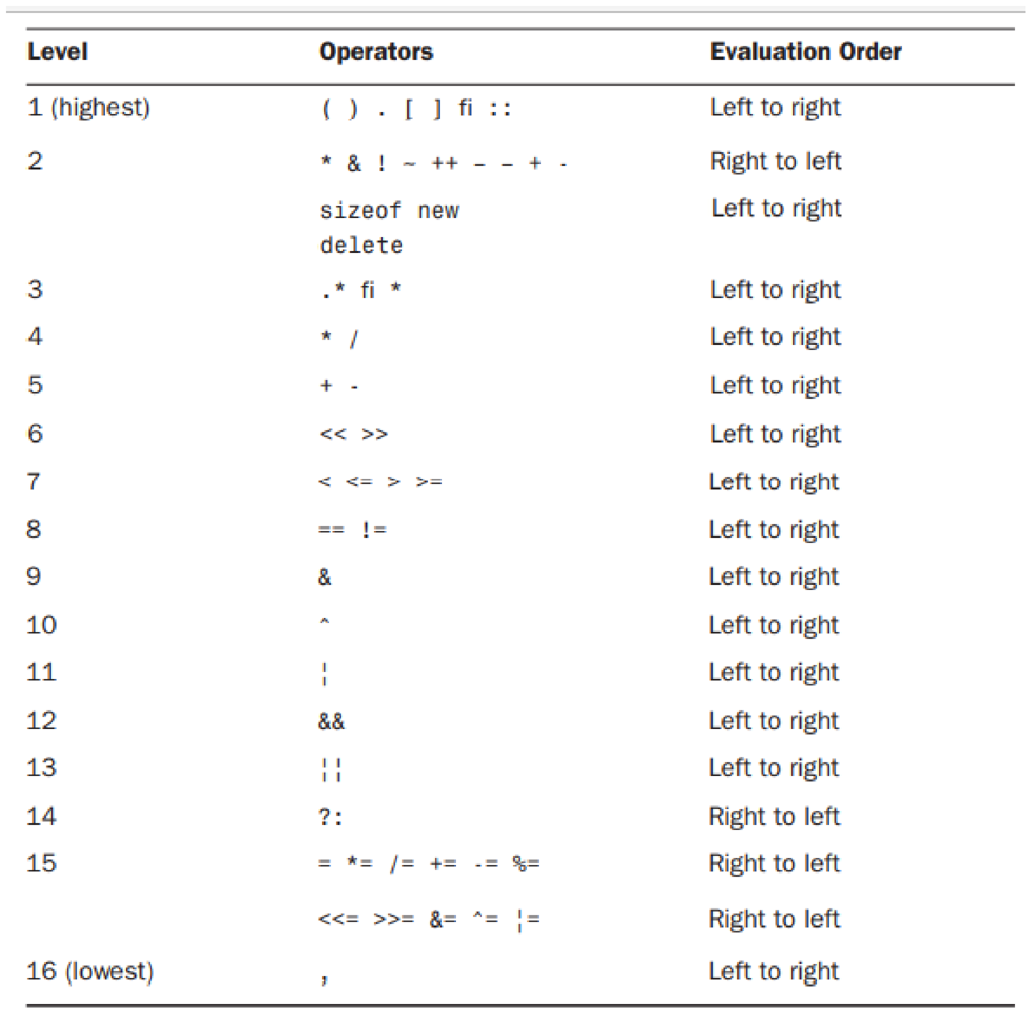
Line 13 produces this output: “The year 2013 passes.” The year remains 2013 because the postfix operator changes the value of year after it is displayed

There are three ways of adding 1 to a variable in C++: a = a + 1, a += 1, and a++. This leads to some confusion about which one is best to use. There’s no best way. As long as you know what your code is doing, all three ways are perfectly acceptable.

Operator Precedence
The values produced by complex expressions depend on the order of precedence, which is the order in which expressions are evaluated. Here’s a complex expression with three operators: int x = 5 + 3 * 8;

This expression sets x to 64 if addition takes place before multiplication, because 8 times 8 equals 64. If multiplication takes place before addition, x equals 29 because 5 plus 24 equals 29.

Every operator has a precedence value. Multiplication has higher precedence than addition, so the expression sets x to 29. The precedence of operators is shown in Table below:



Level	Operators	Evaluation Order
1 (highest)	`( ) . [ ] fi ::`	Left to right
2	`* & ! ~ ++ – – + -`	Right to left
	`sizeof new delete`	Left to right
3	`.* fi *`	Left to right
4	`* /`	Left to right
5	`+ -`	Left to right
6	`<< >>`	Left to right
7	`< <= > >=`	Left to right
8	`== !=`	Left to right
9	`&`	Left to right
10	`^`	Left to right
11	`¦`	Left to right
12	`&&`	Left to right
13	`¦¦`	Left to right
14	`?:`	Right to left
15	`= *= /= += -= %=`	Right to left
	`<<= >>= &= ^= ¦=`	Right to left
16 (lowest)	`,`	Left to right

You are introduced to most of these operators in later hours. Operators are evaluated from top of the table down. Operators with the same precedence are evaluated from left to right or right to left, as indicated in the table. Looking at the table, you can see that the multiplication operator * and division operator / have higher precedence than the addition operator + and subtraction operator -. For this reason, multiplication and division are handled before addition and subtraction.

When two mathematical operators have the same precedence, they are performed in left-to-right order. Here's an expression with two multiplication operators and three addition operators:
int x = 5 + 3 + 8 * 9 + 6 * 4;

Because multiplication has higher precedence than addition and the same operators have left-to-right order, 8 times 9 is evaluated first and becomes 72: int x = 5 + 3 + 72 + 6 * 4;

Next, 6 times 4 is evaluated: int x = 5 + 3 + 72 + 24;

Now the addition operators are handled in left-to-right order. The final result is that x equals 104.

Some operators, such as assignment, are evaluated in right-to-left order: int z = x = y + 13;

The first expression evaluated is y + 13, which is assigned to x. Next, x is assigned to z.

When precedence order doesn't meet your needs, you can use parentheses to impose a different order. Items within parentheses are evaluated at a higher precedence than any mathematical operators: int totalSeconds = (minutesWork + minutesTravel) * 60;

This expression adds minutesWork and minutesTravel, multiplies the result by 60, and assigns it to totalSeconds.

Parentheses can be nested within each other. The innermost parenthesis are evaluated first: totalSeconds = ((secondsWork * 60) + minutesTravel) * 60;

When in doubt, use parentheses to make an expression's meaning clear. They do not affect a program's performance, so there's no harm in using them even in cases where they wouldn't be needed.

Relational Operators
Relational operators are used for comparisons to determine when two numbers are equal or one is greater or less than the other. Every relational expression returns either true or false. The relational operators are presented in Table below:

Table: The Relational Operators

Name	Operator	Sample	Evaluates
Equals	`==`	`100 == 50;`	false
		`50 == 50;`	true
Not equal	`!=`	`100 != 50;`	true
		`50 != 50;`	false
Greater than	`>`	`100 > 50;`	true
		`50 > 50;`	false
Greater than or equals	`>=`	`100 >= 50;`	true
		`50 >= 50;`	true
Less than	`<`	`100 < 50;`	false
		`50 < 50;`	false
Less than or equals	`<=`	`100 <= 50;`	false
		`50 <= 50;`	true

The Else Clause

A program can execute one statement if an if condition is true and another if it is false. The else keyword identifies the statement to execute when the condition is false:

```
if (zombies == 0)
    std::cout << "No more zombies!\n";
else
    std::cout << "Beware the zombie apocalypse!\n";
```

The Grader program in Listing 4.3 demonstrates the use of conditional statements.

The Full Text of Grader.cpp

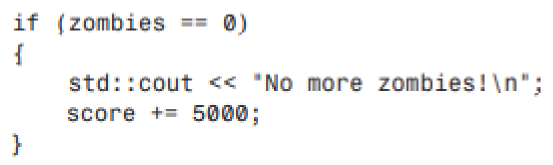
```
#include <iostream>

int main()
{
    int grade;
    std::cout << "Enter a grade (1-100): ";
    std::cin >> grade;

    if (grade >= 70)
        std::cout << "\nPass\n";
    else
        std::cout << "\nFail\n";

    return 0;
}
```

This program uses another part of the input-output library included by the directive in line 1: the std::cin function, which takes a line of user input. Line 6 displays a query to the user: “Enter a grade (1-100).” Line 7 uses std::cin to collect input from the user, storing it in the integer variable grade.

Grader displays different output depending on what the user entered as a grade. This variability employs the if-else conditional in lines 9–12.

Here’s an example of its output: Enter a grade (1-100): 68
Fail

Compound If Statements
Compound statements can be used anywhere in code that a single statement could be placed.
The if and if-else conditionals often are followed by compound statements:



```
if (zombies == 0)
{
     std::cout << "No more zombies!\n";
     score += 5000;
}
```

This code does two things when zombies equal 0: It displays “No more zombies!” and adds 5000 to the variable score. If zombies do not equal 0, neither of these things occurs.

Any statement can be used with an if conditional, including another if conditional. clause, even another if or else statement.

Consider the grading system; write a C++ program to calculate grades:

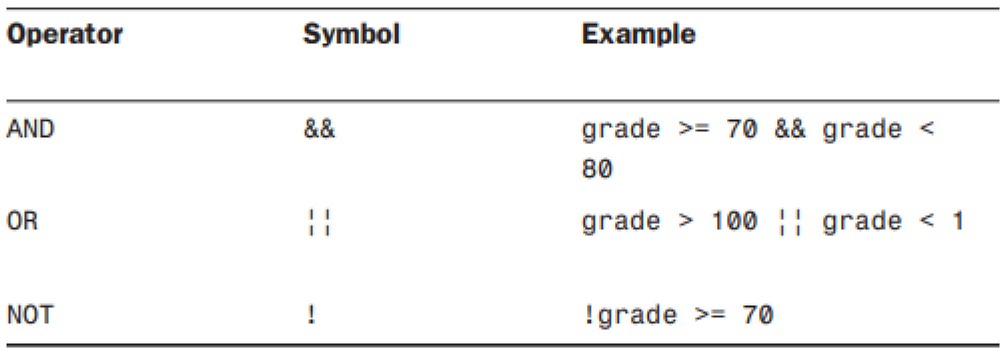
100 – 90	A+
90 – 80	A
80 – 70	B+
70 – 60	B
60 – 50	C
50 – 40	D
40 – 30	E
30 – 0	F

The Full Text of Grade.cpp

```
#include <iostream>
using namespace std;

int main(){
    int marks;
    cout<<"Enter Your Marks: ";
    cin>>marks;
    if (marks >= 90){
        cout<<"Your Grade is A+";
    }
    else if (marks >= 80){
        cout<<"Your Grade is A";
    }
    else if (marks >= 70){
        cout<<"Your Grade is B+";
    }
    else if (marks >= 60){
        cout<<"Your Grade is B";
    }
    else if (marks >= 50){
        cout<<"Your Grade is C";
    }
    else if (marks >= 40){
        cout<<"Your Grade is D";
    }
    else if (marks >= 30){
        cout<<"Your Grade is E";
    }
    else if (marks <= 30){
        cout<<"Your Grade is F";
    }
    else{
        cout<<"Enter Valid Marks";
    }
    return 0;
}
```

Logical Operators

The if-else conditionals used so far have a single expression as the condition. It's possible to test more than one condition using the logical operators **&&** (also called AND) and **||** (OR). The logical operator **!** (NOT) tests whether an expression is false. These operators are listed in Table below:



Operator	Symbol	Example
AND	&&	`grade >= 70 && grade < 80`
OR	¦¦	`grade > 100 ¦¦ grade < 1`
NOT	!	`!grade >= 70`

Relational Precedence

Relational operators and logical operators, like other operators, return a value of true or false and have a precedence order that determines which relations are evaluated first. This fact is important when determining the value of the statement such as the following: if (x > 5 && y > 5 || z > 5)

The logical AND and OR operators have the same precedence, so they are evaluated in left-to-right order. For this expression to be true, both x and y must be greater than 5 or z must be greater than 5. Parentheses can be used to impose a different order: if (x > 5 && (y > 5 || z > 5))