Topics in this Note Set
- Arithmetic Operators
- Order of operations
- Relational, and Logical Operators
- XOR Encryption
- Algorithms: Thinking Like a Programmer
- Selection Statements
- Repetition Statements

Arithmetic Operators
C++ features the standard arithmetic operators: +, -, *, and /
- The “*” represents multiplication
- The “/” represents division

Integer Division
A word about division:
- when you divide an int by an int, you get an int
- for example, 6 / 4 = 1, not 1.5

Another example:
```cpp
int x = 6;
int y = 4;
cout << x / y << endl;
```
This prints out 1, not 1.5.

How do we fix this? There are a couple of ways.
1. Declare one of the variables as a double, because an int divided by a double (or vice versa) is a double.
```cpp
int x = 6;
double y = 4;
cout << x / y << endl;
```
2. Use static_cast to force the computer to consider one of the terms a double instead of an int.
```cpp
int x = 6;
int y = 4;
cout << x / static_cast<double>(y) << endl;
```

Order of Operations
There are some simple rules when it comes to evaluating arithmetic statements in C++:
- anything in parentheses is done first
- then multiplication and division left to right
- then addition and subtraction left to right
- parentheses rule!

Examples: To what does each expression evaluate:
1. 4 + 3 * 5 – 3
2. 5 / 4 + 3 * 6
3. \((2 + 5) \times 4 / (2 + 2)\)

4. int x = 4, y = 3;
   double a = 2.0;
   double z = x + (y + a) / a;
   cout << z << endl;

**Character Data Type and Arithmetic**

There is a data type called char. A char is actually a numeric data type, but you typically use it to store a single character. The numeric value of these variables is the ASCII code for the character.

Character literals are expressed in single quotes. Contrast this with strings, which are multiple characters enclosed in double quotes.

Here is an example:
   char ch1 = 'A';
   char ch2 = ch1 + 1;
   cout << ch2;

This will output the character B because the ASCII code for B is one more than the ASCII code for A.

**Accessing Individual Characters in a String**

You can access individual characters in a string using \[\text{#}\] next to the string variable’s name. The \#\ represents the position of the character in which you are interested. \#\ can range from 0 to one less than the number of characters in the string.

**A Simple Encryption Example**

Here’s some code that takes the word “Howdy” and advances each character in the word by 1.

```c++
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string greeting = "howdy";
    greeting[0] = greeting[0] + 1;
    cout << greeting << endl;
    system("pause");
    return 0;
}
```

**Relational & Logical Operators**

Relational operators are called relational operators because they compare one value to another. These include >, <, >=, <=, == (for equality), and != (for not equal).

Relational operators are used in boolean expression. A Boolean expression is a mathematical sentence that returns true or false.
For example, if we have the following declaration:

```c++
int x = 4;
```

then the expressions

```c++
(x > 2)
```

is a Boolean expression that returns true.

These Boolean expressions are often joined together to form compound statements. The operators that join them are called logical operators.

- `&&` means and
- `||` means or
- `!` means not

We saw those in the preceding note set.

For example, if we have the following declaration:

```c++
int x = 4, y = 7;
```

then this is a Boolean expression:

```c++
(x <= 4) && (y > 3)
```

Its value is true because both x >= 4 and y > 3.

There is one other logical operator that is interesting:

- `^` means XOR

XOR is like OR except that the result is true if and only if 1 but not both logical expressions are true.

**XOR Encryption**

XOR is at the heart of another encryption mechanism. Here is the code:

```c++
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str = "Hey";
    string key = "xyz";
    str[0] = str[0] ^ key[0];
    cout << str << " " << endl;
    str[0] = str[0] ^ key[0];
    cout << str << " " << endl;
    system("pause");
    return 0;
}
```

**Algorithms: Thinking Like a Programmer**

Up to now, we've gotten a little experience writing C++ and Java programs. These were simple programs, but it was still pretty cool to get something running. What about not-so-simple programs? How do we go about writing those?

We need to learn how to think like a programmer.
**Thinking Like a Programmer**

Writing a program is like writing an instruction manual that the computer is supposed to read and execute.

Except that the computer is just about the dumbest thing on the face of the planet. You have to spell out every little minute detail.

After all, computers are good only at
- inputting and outputting information
- performing arithmetic and logic
- performing comparisons between values

Beyond that, the computer needs some help.

Being able to write a program takes practice.

Is there a standard approach that can make the task of writing programs simpler?
Yes. The key is to write an algorithm.

**Algorithm**

- Is a detailed, step-by-step recipe for solving a problem
- If you write the algorithm well, writing the code is easy

In other words, if you do a good enough job in advance, the coding becomes a fairly straightforward endeavor.

How do you come up with an algorithm?

Ask yourself three questions:
- What does the program need to do?
- For what information does the program need to ask?
- What information does the program need to show?

**As you answer these questions, you devise an algorithm!**

The algorithm is a step-by-step listing of instructions / steps that spell out in gruesome detail how to solve a problem.

We’ll now take a look at several examples.

**Example #1**
Write an algorithm (a plan) for finding the perimeter and area of a rectangle.
Example #2
Write an algorithm for converting a temperature between fahrenheit and celsius scales. The user will enter a temperature in either fahrenheit or celsius, and your proposed program must tell him or her what the temperature is in the other temperature scale.

Example #3
A salesperson leaves his home every Monday and returns every Friday. He travels by company car. Each day on the road, the salesperson records the amount of gasoline put in the car. Given the starting odometer reading (that is, the odometer reading before he leaves Monday) and the ending odometer reading (the odometer reading after he returns home on Friday), design an algorithm to find the average miles per gallon. Sample data is as follows:

68723 71289 15.75 16.30 10.95 20.65 30.00
Example #4
Acme University computes pay as follows:
- Base pay
- $1000 for every year with Acme
- $5000 for Masters + $10000 for Ph.D.

Write an algorithm to compute the wage for an employee of Acme

Divide and Conquer
More complicated problems are best solved by breaking them into sub-problems
This is called Top-Down Design

Example #5
A teacher has been receiving many questions from students about grades lately. A student comes in and wants to know from the teacher what her grade in the class is right now. The teacher wants to write a program that will give him the answer quickly. Design an algorithm for the teacher’s program that will allow him to compute the grade for a student simply by entering the name of the student. Note that grades are stored in a file on the teacher’s hard drive.

Selection Statements
A selection statement is one that selects among one or more different courses of action depending on the value of a boolean expression.

There are several varieties of selection statements:

- if
- if – else
- if – else if – else if - ... – else
- switch

Let’s look at these.

```java
if (boolean_expression)
    single_statement;
```

Always acceptable to use braces, even when you have just a single statement

```java
if (boolean_expression)
{
    statement_1;
    statement_2;
    ...
```
statement_n;
}

if-else
if (boolean_expression)
{
    1_or_more_statements;
}
else
{
    1_or_more_statements;
}

if (boolean_expression)
{
    1_or_more_statements;
} else if (boolean_expression)
{
    1_or_more_statements;
} else
{
    1_or_more_statements;
}

switch
switch (int_type_expression)
{
    case value1:
        statement1;
        statement2;
        ...
        break;
    case value2:
        statement1;
        statement2;
        ...
        break;
    default:
        statement1;
        statement2;
        ...
}

Using braces helps prevent the “dangling-else” problem.
else’s always go with the immediately preceding if statement unless there are braces.

Without the break statements, you keep going from the first condition that was true.

Example
Write a program called Compare that inputs two numbers from the user and writes a statement saying whether the first number is less than, greater than, or equal to the second number and also tests and reports if the product of the two numbers is divisible by 2, 3, or 5.
Repetition Statements
So far we have seen 2 different flows of control in programs:
- sequence
  - one statement after another
- selection
  - choose between multiple different sets of instructions based on the value of a boolean condition

The last pattern that we typically see is repetition
  - doing something over and over until a condition is no longer met

There are several different repetition statements:
- while
- do ... while
- for

The first two, while and do/while, are useful for either sentinel or counter controlled loops, while for can be used only for counter-controlled loops
- A **counter-controlled loop** is one for which you can predict ahead of time how many times it will execute.
- A **sentinel-controlled loop** is one for which you can’t predict ahead of time how many times it will execute.
  - for example, have the user enter positive numbers. You know they are done when the enter a negative number or 0. The user may enter 1 number or 500; you don’t know ahead of time.

Let’s survey these different loops:

```plaintext
while (boolean_expression)
{
  1_or_more_statements;
}
```

```plaintext
do ...
{
```
while (boolean_expression);

for (counter_expression;
     continuation_condition;
     counter_adjustment)
{
    1_or_more_statements;
}

break and continue
break leaves a loop completely
continue just skips the rest of the statements in a loop,
but starts again at the next iteration of the loop

Example Write a program that sums up the numbers from 1 to 100.

Example Write an application that computes the sum of the numbers entered by
the user. Stop accepting numbers when the user indicates that they are
done.