```
//===========================
// Section 8 - Statements and Operators
//===========================

// Expressions
The most basic building block of a program
It's a sequence of operators and operands that specifies a computation
Computes a value from a number of operands

// Examples
34              //Literal
favNumber       //Variable
1.5 + 2.8       // Addition
2 * 5           // Multiplication
a > b           // Relational
a =             // Assignment

// Statements
Complete line f code that performs som action
Usually terminated with a semicolon
Usually contains expressions
many types of statements -
- expression
- null
- compound
- selection
- iteration
- declaration
- jump
- try blocks
- etc

int x;              // Declarations
favNumber = 8;      // assignment
1.5 + 2.8           // expression
x = 2 * 5;          // assignment (with an expression in it)
if (a > b)
 cout << "a > b";   // if statements

// Using operators

C++ has a rich set of operators
Unary       Operates on one operand
            Unary - operand that just negates its operand
binary      Operates on two operands
ternary     Operates on three operands (conditional operator)

Common operaands are grouped as follows
- assignment
- arithmetic
- increment\decrement
- relational
- logical
- member access
- other

// Assignment operator
lhs = rhs
The value of the rhs is stored to the lhs
the value of the rhs must be type compatible with the lhs
the lhs must be assignable
Assignment expression is evaluated to what was just assigned.
More than one variable can be assigned in a single statement
The expression statement evaluates to what we just assigned
- this means it can be chained
```

```
NOTE: Initialization and assignment are different
int myNumber{1}; // This is initialization
myNumber = 100; // This is assignment

l-value     // Location of that variable
r-value     // Contents of some variable

// Chaining them together
num1 = num2 = 100;

// Arithmetic operators
+       // Addition
-       // Subtraction
*       // Multiplication
//      // Division
%       // Modulo (only works with integers)

// When doing integer division, anything after the decimal is truncated

std::cout << 5/10 << std::endl;
- This will return a 0 because it truncates the .5
std::cout << 5.0/10.0 <<std::endl;
- C++ knows these are doubles so you get the correct answer

// Increment and decrement operators
++      Add one
--      subtract one

++num1  prefix - add one then use the value
num++   postfix - use the value then add one

result = ++counter; // Result will get the counter with the 1 added
result = counter++; // Result will get counter without the added 1

NOTE - When postfix applied, counter's value will not be increased until the whole statement has been evaluated

//Mixed expressions and conversions

Mixed type expression
C++ operations can occur on the same type operands
If they are different types, C++ will convert one of them
This could affect calculation results
C++ will automatically try to convert(coercion)
If it can't the compiler complains

Conversions:
Higher (bigger values) vs Lower types (smaller versions) are based on the size of the values the type can hold
- long double, double, float, unsigned long, long, unsigned int, int
- short and char types are always converted to int

Type Coercion: coversion of one operand to another data type
Promotion: conversion to a higher type
- Used in mathematical expressions

Demotion: Conversion to a lower type
- Used with assignment to lower type

// Examples
lower   op  higher
2       *   5.2
The 2 is promoted t 2.0 (a double)

int num{0};
lower   =   higher
num     =   100.2
100.2 will be demoted to an integer
```

```
These are examples of compiler's automatic conversion

// Casting explicitly
int totalAmount{100};
int totalNumber{8};
double average{0.0};

average = totalAmount / totalNumber;
std::cout << average << std::endl; // displays 12

average = static_cast<double>(totalAmount) / totalNumber;
std::cout << average << std::endl; // This will display 12.5
NOTE - if we convert one of the operands to a double the compiler will automatically convert the other

// Older C code looked like this
average = (double)total / count;

Use the new static_cast because it does some extra checking whether it can even be converted

// Testing for equality
== and !=
Compares values of 2 expressions
Evaluates to a boolean true or false value
Commonly used in control flow statements

expression1 == expression2
expression1 != expression2

std::cout << std::boolalpha; // Will turn on true\false for booleans instead of 0\1
std::cout << std::noboolalpha; // Will turn it back off_type

NOTE: Be aware that doubles are stored as approximations so if you test whether
12.0 and 11.99999999999999 are equal, you'll get a true result
You wouldn't use built in doubles in applications that have to have critical presicion

// Relational Operators
Expresion1          OP          Expression2

Operator        Meaning
>               Greater than
>=              Greater than or equal to
<               Less than
<=              Less than or equal to
<=>             Three way (in C++ 20)
It will evaluate to zero if they are equal,
a value less than 0 if the left hand side is greater,
greater than 0 if the right hand side is greater

// Logical operators
Operator        Meaning
--------------------------------
not
!               Negation
and
&&              logical and
or
||              logical or

not is just negation (unary) - truth is opposite of original value
and is only true when both operands are true
or is only false when both operands are false
```

```
//Precedence:
not > and > or
not is a unary operator
and and or are binary operators

// Examples
num1 >= 10 && num1 < 20;
!itsRaining && temperature > 32.0;
isRaining || isSnowing;
temperature > 100 && isHumid || isRaining;

// Short circuit evaluation
C++ stops evaluating as soon as it knows the result
ex1 && ex2 && ex3 // if the first one is already false, the result must be false
ex1 || ex2 || ex3 // if the first one is already true, the result must be true

// Compound assignment operators
Operator        Example             Meaning
+=              lhs += rhs;         lhs = lhs + (rhs);
-=              lhs -= rhs;         lhs = lhs - (rhs);
*=              lhs *= rhs;         lhs = lhs * (rhs);
/=              lhs /= rhs;         lhs = lhs / (rhs);
%=              lhs %= rhs;         lhs = lhs % (rhs);
Bitwise operators below here - used to manipulate bits
--------------------------------------------------------
>>=             lhs >>= rhs;            lhs = lhs >> (rhs);
<<=             lhs <<= rhs;            lhs = lhs << (rhs);
&=              lhs & rhs;          lhs = lhs & (rhs);
^=              lhs ^= rhs;         lhs = lhs ^ (rhs);
|=              lhs |= rhs;         lhs = lhs | (rhs);

a += 1;         //a = a + 1
a /= 5;         // a= a / 5;
a *= b + c;     // a = a * (b + c);

// Operator precedence
Higher to lower (operators on the same row have the same precedence)
Operator                                            Associativity
-------------------------------------------------------------------
[] -> . ()                                          left to right
++ -- not -(unary) *(dereference) & sizeof          right to left
* / %                                               left to right
+ -                                                 left to right
<< >>                                               left to right
< <= > >=                                           left to right
== !=                                               left to right
&                                                   left to right
^                                                   left to right
|                                                   left to right
&&                                                  left to right
||                                                  left to right
= op= ?:                                            right to left

// What is associativity
- Use precedence rules when adjacent operators are different
- Use associativity rules when adjacent operators have the same precedence
- Evaluate the expression in the direction indicated
- Use parentheses to remove any doubt
```

```cpp
//===========================
// Section 8 Challenge - without modulo operator
//===========================

#include <iostream>
// Version without using modulo
int main()
{
    // Ask the user to enter the number of cents
    // Display how to provide the change
    // 1 dollar is 100 cents
    // 1 quarter is 25 cents
    // 1 dime is 10 cents
    // 1 nickel is 5 cents
    // 1 penny is 1 cent

    int numberOfDollars{};
    int numberOfQuarters{};
    int numberOfDimes{};
    int numberOfNickels{};
    int numberOfPennies{};
    int swap{};

    int dollaramount{ 100 };
    int quarteramount{ 25 };
    int dimeamount{ 10 };
    int nickelamount{ 5 };
    int pennyamount{ 1 };
    int change{};
    std::cout << "Enter the number of cents to make change for: ";
    std::cin >> change;

    swap = change / dollaramount;
    numberOfDollars = swap;
    change = change - dollaramount * swap;

    swap = change / quarteramount;
    numberOfQuarters = swap;
    change = change - quarteramount * swap;

    swap = change / dimeamount;
    numberOfDimes = swap;
    change = change - dimeamount * swap;

    swap = change / nickelamount;
    numberOfNickels = swap;
    change = change - nickelamount * swap;

    swap = change / pennyamount;
    numberOfPennies = swap;

    std::cout << "Number of  dollars: " << numberOfDollars << std::endl;
    std::cout << "Number of  quarters: " << numberOfQuarters << std::endl;
    std::cout << "Number of  dimes: " << numberOfDimes << std::endl;
    std::cout << "Number of  nickels: " << numberOfNickels << std::endl;
    std::cout << "Number of  pennies: " << numberOfPennies << std::endl;

}
//===========================
// Section 8 Challenge - without modulo operator
//===========================
//===========================
// Section 8 Challenge - with modulo operator
//===========================

#include <iostream>

// version with modulo operator
int main() {
    // Ask the user to enter the number of cents
    // Display how to provide the change
    // 1 dollar is 100 cents
    // 1 quarter is 25 cents
    // 1 dime is 10 cents
    // 1 nickel is 5 cents
    // 1 penny is 1 cent

    int numberofdollars{};
    int numberofquarters{};
    int numberofdimes{};
    int numberofnickels{};
    int numberofpennies{};
    int change{};
    int swap{};
    int dollarAmount{ 100 };
    int quarterAmount{ 25 };
    int dimeAmount{ 10 };
    int nickelAmount{ 5 };
    int pennyAmount{ 1 };
    std::cout << "enter the number of cents to make change for: ";
    std::cin >> change;

    std::cout << change % 100 << std::endl;

    numberofdollars = change / dollarAmount;
    change %= 100;
    numberofquarters = change / quarterAmount;
    change %= 25;
    numberofdimes = change / dimeAmount;
    change %= 10;
    numberofnickels = change / nickelAmount;
    change %= 5;
    numberofpennies = change / pennyAmount;

    std::cout << "number of  dollars: " << numberofdollars << std::endl;
    std::cout << "number of  quarters: " << numberofquarters << std::endl;
    std::cout << "number of  dimes: " << numberofdimes << std::endl;
    std::cout << "number of  nickels: " << numberofnickels << std::endl;
    std::cout << "number of  pennies: " << numberofpennies << std::endl;
}
//===========================
// Section 8 Challenge - with modulo operator
//===========================
```