

```

1 //=====
2 // Section 8 - Statements and Operators
3 //=====
4
5 // Expressions
6 The most basic building block of a program
7 It's a sequence of operators and operands that specifies a computation
8 Computes a value from a number of operands
9
10 // Examples
11 34 //Literal
12 favNumber //Variable
13 1.5 + 2.8 // Addition
14 2 * 5 // Multiplication
15 a > b // Relational
16 a = // Assignment
17
18 // Statements
19 Complete line of code that performs some action
20 Usually terminated with a semicolon
21 Usually contains expressions
22 many types of statements -
23 - expression
24 - null
25 - compound
26 - selection
27 - iteration
28 - declaration
29 - jump
30 - try blocks
31 - etc
32
33 int x; // Declarations
34 favNumber = 8; // assignment
35 1.5 + 2.8 // expression
36 x = 2 * 5; // assignment (with an expression in it)
37 if (a > b)
38     cout << "a > b"; // if statements
39
40 // Using operators
41
42 C++ has a rich set of operators
43 Unary Operates on one operand
44 Unary - operand that just negates its operand
45 binary Operates on two operands
46 ternary Operates on three operands (conditional operator)
47
48 Common operands are grouped as follows
49 - assignment
50 - arithmetic
51 - increment\decrement
52 - relational
53 - logical
54 - member access
55 - other
56
57 // Assignment operator
58 lhs = rhs
59 The value of the rhs is stored to the lhs
60 the value of the rhs must be type compatible with the lhs
61 the lhs must be assignable
62 Assignment expression is evaluated to what was just assigned.
63 More than one variable can be assigned in a single statement
64 The expression statement evaluates to what we just assigned
65 - this means it can be chained
66
67
68
69

```

```

70 NOTE: Initialization and assignment are different
71 int myNumber{1}; // This is initialization
72 myNumber = 100; // This is assignment
73
74 l-value      // Location of that variable
75 r-value      // Contents of some variable
76
77 // Chaining them together
78 num1 = num2 = 100;
79
80 // Arithmetic operators
81 +           // Addition
82 -           // Subtraction
83 *           // Multiplication
84 //          // Division
85 %           // Modulo (only works with integers)
86
87 // When doing integer division, anything after the decimal is truncated
88
89 std::cout << 5/10 << std::endl;
90 - This will return a 0 because it truncates the .5
91 std::cout << 5.0/10.0 <<std::endl;
92 - C++ knows these are doubles so you get the correct answer
93
94 // Increment and decrement operators
95 ++          Add one
96 --          subtract one
97
98 ++num1      prefix - add one then use the value
99 num++       postfix - use the value then add one
100
101 result = ++counter; // Result will get the counter with the 1 added
102 result = counter++; // Result will get counter without the added 1
103
104 NOTE - When postfix applied, counter's value will not be increased until the whole
        statement has been evaluated
105
106 //Mixed expressions and conversions
107
108 Mixed type expression
109 C++ operations can occur on the same type operands
110 If they are different types, C++ will convert one of them
111 This could affect calculation results
112 C++ will automatically try to convert (coercion)
113 If it can't the compiler complains
114
115 Conversions:
116 Higher (bigger values) vs Lower types (smaller versions) are based on the size of the
        values the type can hold
117 - long double, double, float, unsigned long, long, unsigned int, int
118 - short and char types are always converted to int
119
120 Type Coercion: conversion of one operand to another data type
121 Promotion: conversion to a higher type
122 - Used in mathematical expressions
123
124 Demotion: Conversion to a lower type
125 - Used with assignment to lower type
126
127 // Examples
128 lower    op    higher
129 2        *    5.2
130 The 2 is promoted to 2.0 (a double)
131
132 int num{0};
133 lower    =    higher
134 num      =    100.2
135 100.2 will be demoted to an integer
136

```

```

137 These are examples of compiler's automatic conversion
138
139 // Casting explicitly
140 int totalAmount{100};
141 int totalNumber{8};
142 double average{0.0};
143
144 average = totalAmount / totalNumber;
145 std::cout << average << std::endl; // displays 12
146
147 average = static_cast<double>(totalAmount) / totalNumber;
148 std::cout << average << std::endl; // This will display 12.5
149 NOTE - if we convert one of the operands to a double the compiler will automatically
      convert the other
150
151 // Older C code looked like this
152 average = (double)total / count;
153
154 Use the new static_cast because it does some extra checking whether it can even be
      converted
155
156 // Testing for equality
157 == and !=
158 Compares values of 2 expressions
159 Evaluates to a boolean true or false value
160 Commonly used in control flow statements
161
162 expression1 == expression2
163 expression1 != expression2
164
165 std::cout << std::boolalpha; // Will turn on true\false for booleans instead of 0\1
166 std::cout << std::noboolalpha; // Will turn it back off_type
167
168 NOTE: Be aware that doubles are stored as approximations so if you test whether
169 12.0 and 11.999999999999999 are equal, you'll get a true result
170 You wouldn't use built in doubles in applications that have to have critical precision
171
172 // Relational Operators
173 Expression1      OP      Expression2
174
175 Operator      Meaning
176 >             Greater than
177 >=            Greater than or equal to
178 <             Less than
179 <=            Less than or equal to
180 <=>           Three way (in C++ 20)
181 It will evaluate to zero if they are equal,
182 a value less than 0 if the left hand side is greater,
183 greater than 0 if the right hand side is greater
184
185 // Logical operators
186 Operator      Meaning
187 -----
188 not
189 !             Negation
190 and
191 &&            logical and
192 or
193 ||            logical or
194
195 not is just negation (unary) - truth is opposite of original value
196 and is only true when both operands are true
197 or is only false when both operands are false

```

```

204 //Precedence:
205 not > and > or
206 not is a unary operator
207 and and or are binary operators
208
209 // Examples
210 num1 >= 10 && num1 < 20;
211 !itsRaining && temperature > 32.0;
212 isRaining || isSnowing;
213 temperature > 100 && isHumid || isRaining;
214
215 // Short circuit evaluation
216 C++ stops evaluating as soon as it knows the result
217 ex1 && ex2 && ex3 // if the first one is already false, the result must be false
218 ex1 || ex2 || ex3 // if the first one is already true, the result must be true
219
220 // Compound assignment operators
221 Operator      Example      Meaning
222 +=            lhs += rhs;    lhs = lhs + (rhs);
223 -=            lhs -= rhs;    lhs = lhs - (rhs);
224 *=            lhs *= rhs;    lhs = lhs * (rhs);
225 /=            lhs /= rhs;    lhs = lhs / (rhs);
226 %=            lhs %= rhs;    lhs = lhs % (rhs);
227 Bitwise operators below here - used to manipulate bits
228 -----
229 >>=           lhs >>= rhs;    lhs = lhs >> (rhs);
230 <<=           lhs <<= rhs;    lhs = lhs << (rhs);
231 &=            lhs & rhs;      lhs = lhs & (rhs);
232 ^=            lhs ^= rhs;      lhs = lhs ^ (rhs);
233 |=            lhs |= rhs;      lhs = lhs | (rhs);
234
235 a += 1;        //a = a + 1
236 a /= 5;        // a= a / 5;
237 a *= b + c;    // a = a * (b + c);
238
239 // Operator precedence
240 Higher to lower (operators on the same row have the same precedence)
241 Operator      Associativity
242 -----
243 [] -> . ()     left to right
244 ++ -- not -(unary) *(dereference) & sizeof right to left
245 * / %          left to right
246 + -            left to right
247 << >>          left to right
248 < <= > >=     left to right
249 == !=         left to right
250 &              left to right
251 ^              left to right
252 |              left to right
253 &&             left to right
254 ||            left to right
255 = op= ?:      right to left
256
257 // What is associativity
258 - Use precedence rules when adjacent operators are different
259 - Use associativity rules when adjacent operators have the same precedence
260 - Evaluate the expression in the direction indicated
261 - Use parentheses to remove any doubt
262
263
264
265
266
267
268
269
270
271
272

```

```

273 //=====
274 // Section 8 Challenge - without modulo operator
275 //=====
276
277 #include <iostream>
278 // Version without using modulo
279 int main()
280 {
281     // Ask the user to enter the number of cents
282     // Display how to provide the change
283     // 1 dollar is 100 cents
284     // 1 quarter is 25 cents
285     // 1 dime is 10 cents
286     // 1 nickel is 5 cents
287     // 1 penny is 1 cent
288
289     int numberOfDollars{};
290     int numberOfQuarters{};
291     int numberOfDimes{};
292     int numberOfNickels{};
293     int numberOfPennies{};
294     int swap{};
295
296     int dollaramount{ 100 };
297     int quarteramount{ 25 };
298     int dimeamount{ 10 };
299     int nickelamount{ 5 };
300     int pennyamount{ 1 };
301     int change{};
302     std::cout << "Enter the number of cents to make change for: ";
303     std::cin >> change;
304
305     swap = change / dollaramount;
306     numberOfDollars = swap;
307     change = change - dollaramount * swap;
308
309     swap = change / quarteramount;
310     numberOfQuarters = swap;
311     change = change - quarteramount * swap;
312
313     swap = change / dimeamount;
314     numberOfDimes = swap;
315     change = change - dimeamount * swap;
316
317     swap = change / nickelamount;
318     numberOfNickels = swap;
319     change = change - nickelamount * swap;
320
321     swap = change / pennyamount;
322     numberOfPennies = swap;
323
324     std::cout << "Number of dollars: " << numberOfDollars << std::endl;
325     std::cout << "Number of quarters: " << numberOfQuarters << std::endl;
326     std::cout << "Number of dimes: " << numberOfDimes << std::endl;
327     std::cout << "Number of nickels: " << numberOfNickels << std::endl;
328     std::cout << "Number of pennies: " << numberOfPennies << std::endl;
329
330 }
331 //=====
332 // Section 8 Challenge - without modulo operator
333 //=====
334
335
336
337
338
339
340
341

```

```

342 //=====
343 // Section 8 Challenge - with modulo operator
344 //=====
345
346 #include <iostream>
347
348 // version with modulo operator
349 int main() {
350     // Ask the user to enter the number of cents
351     // Display how to provide the change
352     // 1 dollar is 100 cents
353     // 1 quarter is 25 cents
354     // 1 dime is 10 cents
355     // 1 nickel is 5 cents
356     // 1 penny is 1 cent
357
358     int numberofdollars{};
359     int numberofquarters{};
360     int numberofdimes{};
361     int numberofnickels{};
362     int numberofpennies{};
363     int change{};
364     int swap{};
365     int dollarAmount{ 100 };
366     int quarterAmount{ 25 };
367     int dimeAmount{ 10 };
368     int nickelAmount{ 5 };
369     int pennyAmount{ 1 };
370     std::cout << "enter the number of cents to make change for: ";
371     std::cin >> change;
372
373     std::cout << change % 100 << std::endl;
374
375     numberofdollars = change / dollarAmount;
376     change %= 100;
377     numberofquarters = change / quarterAmount;
378     change %= 25;
379     numberofdimes = change / dimeAmount;
380     change %= 10;
381     numberofnickels = change / nickelAmount;
382     change %= 5;
383     numberofpennies = change / pennyAmount;
384
385     std::cout << "number of dollars: " << numberofdollars << std::endl;
386     std::cout << "number of quarters: " << numberofquarters << std::endl;
387     std::cout << "number of dimes: " << numberofdimes << std::endl;
388     std::cout << "number of nickels: " << numberofnickels << std::endl;
389     std::cout << "number of pennies: " << numberofpennies << std::endl;
390 }
391 //=====
392 // Section 8 Challenge - with modulo operator
393 //=====
394
395

```