Operations, math functions and weird integer types

```
/*
    . Topics:
        . Basic operations:
            . Addition, subtraction, multiplication, division, modulus
            . Precedence and associativity
            . Increment and decrement (prefix and postfix)
            . Compound assignment operators (+=, -=, *=, /=, %=)
            . Relational operators (==, ≠, <, >, ⩽, ⩾)
            . Logical operators (&&, ||, !)
            . Math functions
            . Weird integral types
            . Data conversions:
                . Implicit conversions
                . Explicit conversions
                . Overflow and underflow
*/
```

Basic Operations

```rust
// Addition
int number1{2};
int number2{7};

int result = number1 + number2;
fmt::println("addition - result : {}", result);

// Subtraction
result = number2 - number1;
fmt::println("subtraction - result : {}", result);

result = number1 - number2;
fmt::println("subtraction - result : {}", result);

// Multiplication
result = number1 * number2;
fmt::println("subtraction - result : {}", result);

// Division
result = number2 / number1;
fmt::println("division - result : {}", result);

// Modulus
result = number2 % number1; // 7 % 2
fmt::println("modulus - result : {}", result); // 1

result = 31 % 10;
fmt::println("modulus - result : {}", result); // 1
```

Precedence and associativity

```rust
int a {6};
int b {3};
int c {8};
int d {9};
int e {3};
int f {2};
int g {5};

int result = a + b * c -d/e -f + g; //  6 +  24  -   3 - 2 + 5

fmt::println("result : {}", result);

result = a/b*c +d - e + f;  //   16 + 9 - 3 + 2
fmt::println("result : {}", result);

result = (a + b) * c -d/e -f + g; // 72-3-2+5
fmt::println("result () : {}", result);
```

Prefix and postfix increment

```
int value { 5 };

//Increment by one
value = value + 1; //6
fmt::println("The value is : {}", value); // 6

value = 5; // Reset value to 5

//Decrement by one
value = value - 1; // 4
fmt::println("The value is : {}", value); //4
```

Postfix increment and decrement

```
//Reset value to 5
value = 5;

fmt::println("The value is (incrementing) : {}", value++); // 5
fmt::println("The value is : {}", value); // 6

fmt::println("\n");

//Decrement with postfix

//Reset value to 5
value = 5;

fmt::println("The value is (decrementing) : {}", value--); //5
fmt::println("The value is : {}", value); // 4
```

Prefix increment and decrement

```
//Reset value to 5
value = 5;

++value;
fmt::println("The value is (prefix++) : {}", value); // 6

//Reset value to 5
value = 5;
fmt::println("The value is (prefix++ in place) : {}", ++value); // 6

//Prefix : Decrementing

//Reset value to 5;
value = 5;
--value;
fmt::println("The value is (prefix--) : {}", value); // 4

//Reset value to 5;
value = 5;
fmt::println("The value is (prefix-- in place) : {}", --value);//4
```

Compound assignment operator

```rust
int value {45};

fmt::println("The value is : {}", value);

value = value + 5;
//value +=5; // equivalent to value = value + 5
fmt::println("The value is (after +=5) : {}", value); // 50

value -=5; // equivalent to value = value - 5
fmt::println("The value is (after -=5) : {}", value); // 45

value *=2;
fmt::println("The value is (after *=2) : {}", value); // 90

value /= 3;
fmt::println("The value is (after /=3) : {}", value); // 30

value %= 11;
fmt::println("The value is (after %=11) : {}", value);// 8
```

Relational operators

```
int number1 {20};
int number2 {20};

fmt::println("number1 : {}", number1);
fmt::println("number2 : {}", number2);

fmt::println("Comparing variables");

fmt::println("number1 < number2 : {}", (number1 < number2));
fmt::println("number1 ⩽ number2 : {}", (number1 ⩽ number2));
fmt::println("number1 > number2 : {}", (number1 > number2));
fmt::println("number1 ⩾ number2 : {}", (number1 ⩾ number2));
fmt::println("number1 == number2 : {}", (number1 == number2));
fmt::println("number1 ≠ number2 : {}", (number1 ≠ number2));

fmt::println("store comparison result and use it later");

bool result = (number1 == number2);

fmt::println("{} == {} : {}", number1, number2, result);
```

Logical operators: AND, OR

```cpp
bool a{ true };
bool b{ false };
bool c{ true };

fmt::println("a : {}", a);
fmt::println("b : {}", b);
fmt::println("c : {}", c);


// AND : Evaluates to true when all operands are true.
//       A single false operand will drag
//      the entire expression to evaluating false.

fmt::println("Basic AND operations");

fmt::println(" a && b : {}", (a && b));// false
fmt::println(" a && c : {}", (a && c));// true
fmt::println(" a && b && c : {}", (a && b && c));// false


// OR : Evaluates to true when at least one operand true.
//       A single true operand will push
//      the entire expression to evaluating true.
fmt::println("Basic OR operations");
fmt::println(" a || b : {}", (a || b));
fmt::println(" a || c : {}", (a || c));
fmt::println(" a || b || c : {}", (a || b || c));
```

Logical operators: NOT and combinations

```rust
// NOT : Negates whateve operand you put it with
fmt::println("Basic NOT operations");

fmt::println("!a : {}", !a);
fmt::println("!b : {}", !b);
fmt::println("!c : {}", !c);


// Combinations of all these operators
fmt::println("Combining logical operators");
// !(a &&b) || c
fmt::println("!(a &&b) || c : {}", (!(a && b) || c));// true
```

Logical and relational operators combined

```
int d{ 45 };
int e{ 20 };
int f{ 11 };

fmt::println("Relational and logic operations on integers");
fmt::println("d : {}", d);
fmt::println("e : {}", e);
fmt::println("f : {}", f);

fmt::println("(d > e) && (d > f) : {}", ((d > e) && (d > f)));// true
fmt::println("(d==e) || (e ⩽ f ) : {}", ((d == e) || (e ⩽ f)));
fmt::println("(d < e) || (d > f) : {}", ((d < e) || (d > f)));
fmt::println("(f > e) || (d < f) : {}", ((f > e) || (d < f)));
fmt::println("(d > f) && (f ⩽ d) : {}", ((d > f) && (f ⩽ d)));
fmt::println("(d > e) && (d ⩽ f) : {}", ((d > e) && (d ⩽ f)));
fmt::println("(! a) && (d == e) : {}", ((!a) && (d == e)));
fmt::println("(! a) && (d == e) : {}", ((!a) && (d == e)));
```

Math functions

```cpp
double weight{ 7.7 };

// floor
fmt::println("Weight rounded to floor is : {}", std::floor(weight));

// ceil
fmt::println("Weight rounded to ceil is : {}", std::ceil(weight));

// abs
double savings{ -5000 };

fmt::println("Abs of weight is : {}", std::abs(weight));
fmt::println("Abs of savings is : {}", std::abs(savings));

//round. Halfway points are rounded away from 0. 2.5 is rounded to 3 for example
fmt::println("3.654 rounded to : {}", std::round(3.654));
fmt::println("2.5 is rounded to : {}", std::round(2.5));
fmt::println("2.4 is rounded to : {}", std::round(2.4));
fmt::println("-2.4 is rounded to : {}", std::round(-2.4));
// round: type of result
auto result = std::round(-2.4);
fmt::println("Type of rounded -2.4: {}", typeid(result).name());
```

Math functions

```cpp
// exp : f(x) = e ^ x , where e = 2.71828 . Test the result here against a calculator
double exponential = std::exp(10);
fmt::println("The exponential of 10 is : {}", exponential);
// pow
fmt::println("3^4 is : {}", std::pow(3, 4));
fmt::println("9^3 is : {}", std::pow(9, 3));

// log : reverse function of pow. if 2^3 = 8 , log 8 in base 2 = 3.  Log is like asking
//  to which exponent should we elevate 2 to get eight ? Log, by default computes the log
//  in base e. There also is another function which uses base 10 called log10

// Try the reverse operation of  e^4 = 54.59 , it will be log 54.59 in base e = ?
fmt::println("Log ; to get 54.59, you would elevate e to the power of : {}", std::log(54.59));
// log10 , 10 ^ 4 = 10000  , to get 10k , you'd need to elevate 10 to the power of ? , this is log in base 10
fmt::println("To get 10000, you'd need to elevate 10 to the power of : {}",
  std::log10(10000));// 4
// sqrt
fmt::println("The square root of 81 is : {}", std::sqrt(81));
```

Weird integral types

```cpp
short int var1{ 10 };// 2 bytes
short int var2{ 20 };

char var3{ 40 };// 1
char var4{ 50 };

fmt::print("size of var1 : {}\n", sizeof(var1));
fmt::print("size of var2 : {}\n", sizeof(var2));
fmt::print("size of var3 : {}\n", sizeof(var3));
fmt::print("size of var4 : {}\n", sizeof(var4));

auto result1 = var1 + var2;
auto result2 = var3 + var4;

fmt::print("size of result1 : {}\n", sizeof(result1));// 4
fmt::print("size of result2 : {}\n", sizeof(result2));// 4
```

Implicit conversions in expressions

```
//Data conversions
// Implicit data conversions
//         . The compiler applies implicit conversions
//                when types are different in
//             an expression
//         . Conversions are always done from the smallest
//             to the largest type in this case int is
//             transformed to double before the expression
//             is evaluated.Unless we are doing an assignment
```

Implicit conversions in expressions

```cpp
double price { 45.6 };
int units {10};


// In C++, when an operation involves two different numeric types,
// the operand of the smaller type is implicitly converted to the
// larger type before the operation is performed. This is known as type promotion.
// Here, units (an int) is implicitly converted to double before the multiplication,
// so the result of the operation is also a double.
// Therefore, total_price is deduced to be of type double.

auto total_price = price * units; // units will be implicitly converted to double

fmt::println("Total price : {}", total_price);
fmt::println("sizeof total_price : {}", sizeof(total_price));
```

Implicit conversions in assignments

```
// The assignment operation is going to cause an implicit
// narrowing conversion , y is converted to int before assignment
int x;
double y {45.44};
x = y; // double to int
fmt::println("The value of x is : {}", x);
fmt::println("sizeof x : {}", sizeof(x));
```

Explicit data conversions

```cpp
double x { 12.5 };
double y { 34.6};

int sum = x + y;
fmt::println("The sum is: {}", sum);

//Explicity cast : cast then sum up
sum = static_cast<int>(x) + static_cast<int>(y);
fmt::println("The sum is: {}", sum);

//Explicit cast : sum up then cast, same thing as implicit cast
sum =  static_cast<int> (x + y);
fmt::println("Sum up then cast, result: {}", sum);

//Old style C-cast
double PI {3.14};

//int int_pi = (int)(PI);
int int_pi = static_cast<int>(PI);
fmt::println("PI: {}", PI);
fmt::println("int_pi: {}", int_pi);
```

Overflow

```cpp
unsigned char data {250};
fmt::println("unsigned char min: {} and max: {}", std::numeric_limits<unsigned char>::min(),
                                                  std::numeric_limits<unsigned char>::max());
 ++data;
 fmt::println("data : {}", static_cast<int>(data));
 ++data;
 fmt::println("data : {}", static_cast<int>(data));
 ++data;
 fmt::println("data : {}", static_cast<int>(data));
 ++data;
 fmt::println("data : {}", static_cast<int>(data));
 ++data;
 fmt::println("data : {}", static_cast<int>(data));// 255

 ++data;// Overflow
 fmt::println("data overflow : {}", static_cast<int>(data));// 256 → 0

 data = 1;

 --data; //0
 fmt::println("data : {}", static_cast<int>(data));

 --data; // Underflow: 255
 fmt::println("data underflow : {}", static_cast<int>(data));
```