An Introduction and Example Reference to Varidiac Functions & Arity

The arity of a function is defined as its number of arguments or operands. For example, simple operands like addition, AND, multiplication, etc are 2-ary function and are called binary functions as its Latin-based name. Things like the NOT operand (~) are unary (or 1-ary) operators.

But then there are also functions that can take an unspecified number of arguments - such as the sigma function that sums all of the numbers from a lower limit to an upper one, with a function as one of its arguments.

$$\sum_{n=0}^{u} f(n)$$

Here's an example of a simple C++ adder function that just adds all the values given to it as comma seperated values.

```
template <typename... T>
auto adder(T... args) {
    return (... + args);
}
```

As talked about above, the arity of a function can differ - even be variable. But can a function be *0-ary*? Yes. And in fact, a **nullary** example for Mathematics can be a constant! Like π. Now that we're comfortable with the topic of arity, let's write some code with functions of different arity.

Unary Functions (1-ary)

1. Logical NOT

$$\begin{aligned} \text{NOT } n &= -n - 1 \\ &= -(n + 1) \end{aligned}$$

```
long NOT(const long n) {
  return -(n + 1);
}
```

2. Square Root (using Newton's Method)

$$f(x) = x^2 - n$$

$$f'(x) = 2x$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Taking $i = 9$ and approximating initial first value using brute force, we can find $\sqrt{n}$ easily

```
double sqrt(int n) {
  double x0 = 1;
  while (std::pow(x0, 2) - 1 <= n)
    ++x0;
  for (unsigned _ = 0; _ < 10; ++_)
    x0 -= (std::pow(x0, 2) - n) / (x0 * 2);
  return x0;
}
```

3. Sine

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

```
#define ACCURACY 10                       // Variable accuracy. 10 is good enough.
auto sin(const float x) {
  float power, result = 0;
  int factorial, sign = -1;
  for (unsigned n = 1; n <= ACCURACY; n += 2) {
    power = 1;                            // Reset values for new fraction part.
    factorial = 1;
    for (unsigned j = 1; j <= n; ++j) { // Compute factorial as well as pow(x, n).
      power *= x;
      factorial *= j;
    }
    sign *= -1;                           // Change sign;
    result += sign * power / factorial;
  }
  return result;
}
```

4. Factorial

Binary Functions (2-ary)

1. Logical AND

$$x \text{ AND } y = \sum_{n=0}^{\lfloor \log_2 x \rfloor} 2^n \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right)$$

```
auto AND(const int x, const int y) {
  long result = 0;
  int upper = std::floor(std::log2(x)) + 1;
    for (unsigned n = 0; n < upper; ++n) {
      long two_power_n = std::pow(2, n);
      long x_2n_mod_2  = (long)(std::floor(x / two_power_n)) % 2 );
      long y_2n_mod_2  = (long)(std::floor(y / two_power_n)) % 2 );
      result += two_power_n * x_2n_mod_2 * y_2n_mod_2;
  }
  return result;
}
```

2. Logical OR

$$x \text{ OR } y = \sum_{n=0}^{\lfloor \log_2 x \rfloor} 2^n \left(\left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \right] \bmod 2 \right)$$

```
long OR(int x, int y) {
  long result = 0;
  if (y > x)
    std::swap(x, y);
  int upper = std::floor(std::log2(x)) + 1;
  for (unsigned n = 0; n < upper; ++n) {
    long two_power_n = std::pow(2, n);
    long x_2n_mod_2 = (long)(std::floor(x / two_power_n)) % 2;
    long y_2n_mod_2 = (long)(std::floor(y / two_power_n)) % 2;
    auto prod = x_2n_mod_2 * y_2n_mod_2;
    result += two_power_n * (( x_2n_mod_2 + y_2n_mod_2 + prod ) % 2);
  }
  return result;
}
```

3. Distance From Origin

$$
\begin{aligned}
P &= (x_1, y_1) \\
Q &= (x_2, y_2) \\
d(P, Q) &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \\
&= \sqrt{(0 - x_1)^2 + (0 - y_1)^2} \qquad [\text{when Q } = (0, 0)] \\
&= \sqrt{x_1^2 + y_1^2} \\
&= \sqrt{x^2 + y^2}
\end{aligned}
$$

```
double origin_distance(float x, float y) {
  return std::sqrt(
        std::pow(x, 2) + std::pow(y, 2)
  );
}
```

4. Logarithm

$$
\begin{aligned}
b^y &= x \\
\log_b(x) &= y
\end{aligned}
$$

```
uint16_t log2(uint32_t x) {
  uint16_t result = -1;
  while (x) {
    result++;
    x >>= 1;
  }
  return result;
}

uint32_t log(uint32_t x, uint32_t b) {
  return log2(x) / log2(b);
}
```