Functions

Functions : a roadmap

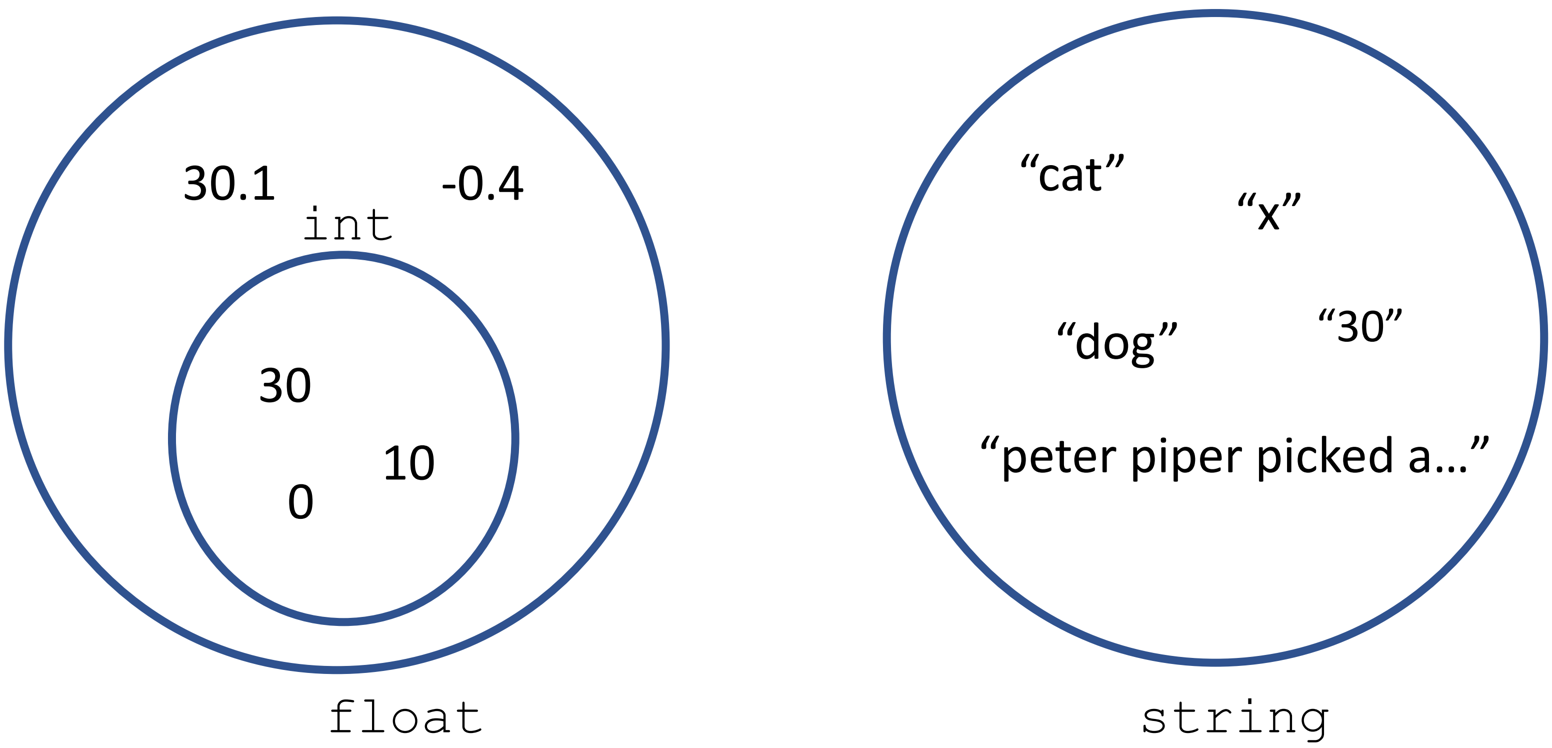
- What we’re going to look at today
  - Mathematical functions
  - Functions in C++
- We will highlight the maths <-> code relationship
  - You’re all mostly familiar with continuous maths*
  - Maths is the original model for functions in code
- Next lecture: heterogenous types and scopes
- Next week : recursion

* The UK English abbreviation: *mathematics* -> *maths*. https://www.youtube.com/watch?v=SbZCECvoaTA

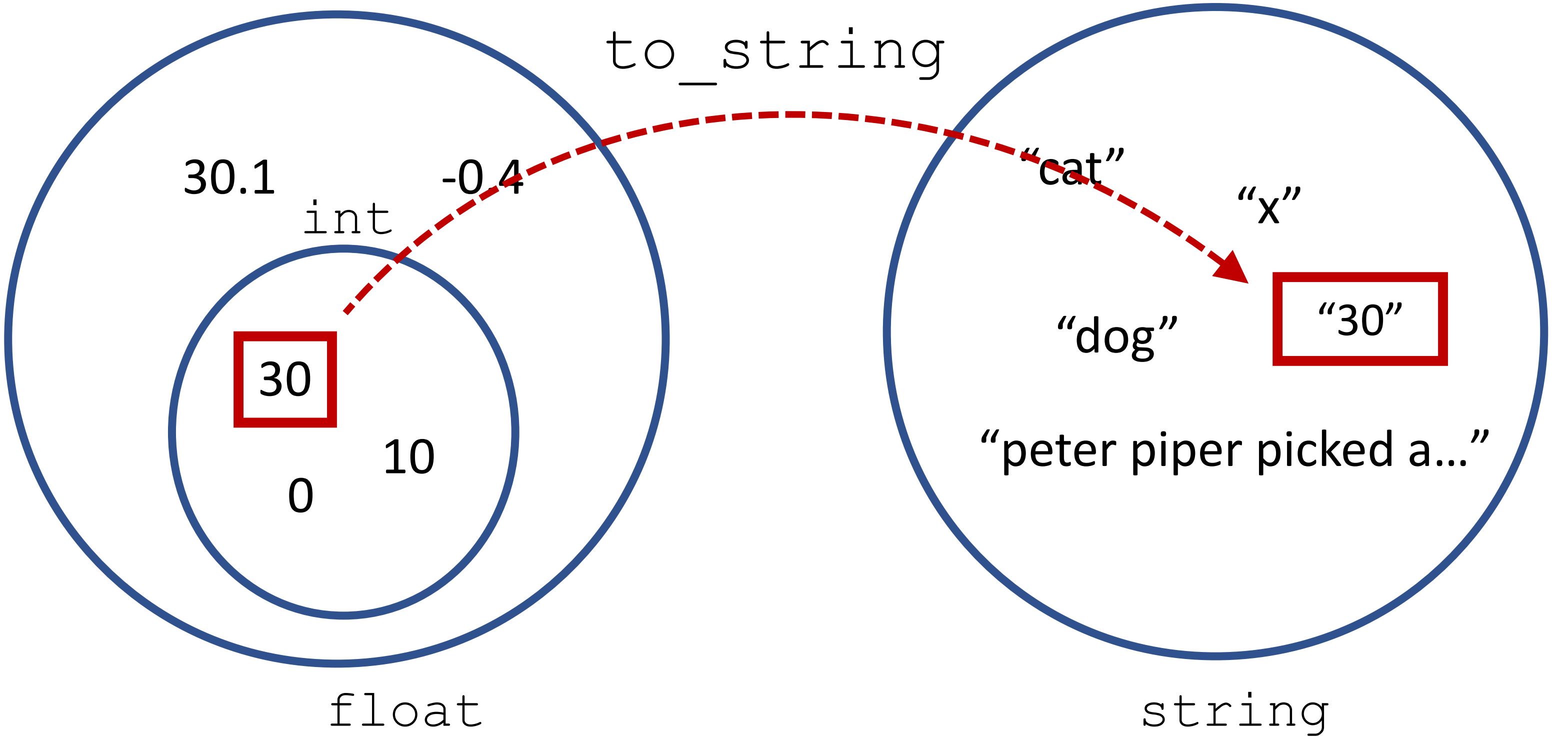
Mathematical functions

Or: what is a function?

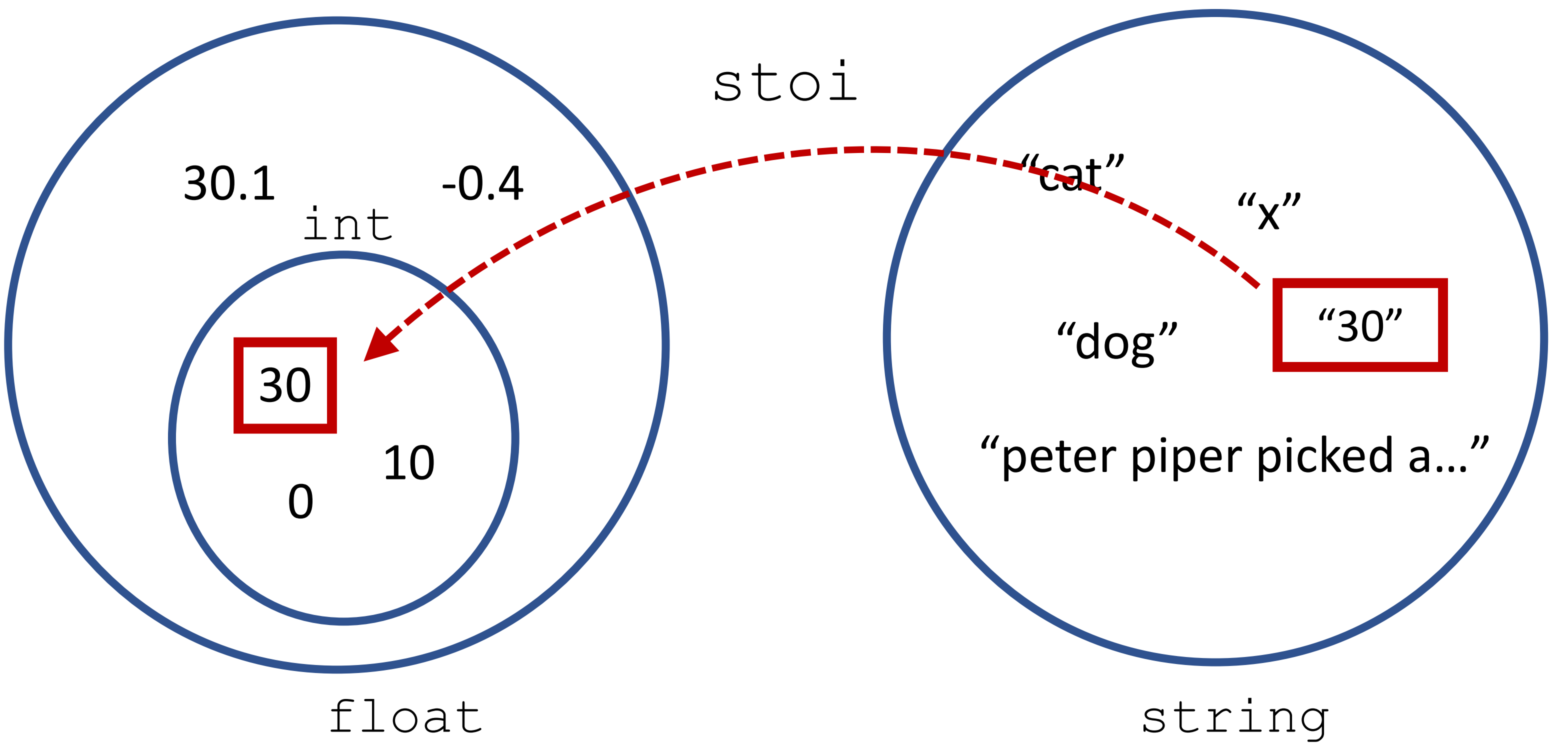
Functions : maps between types



Functions : maps between types

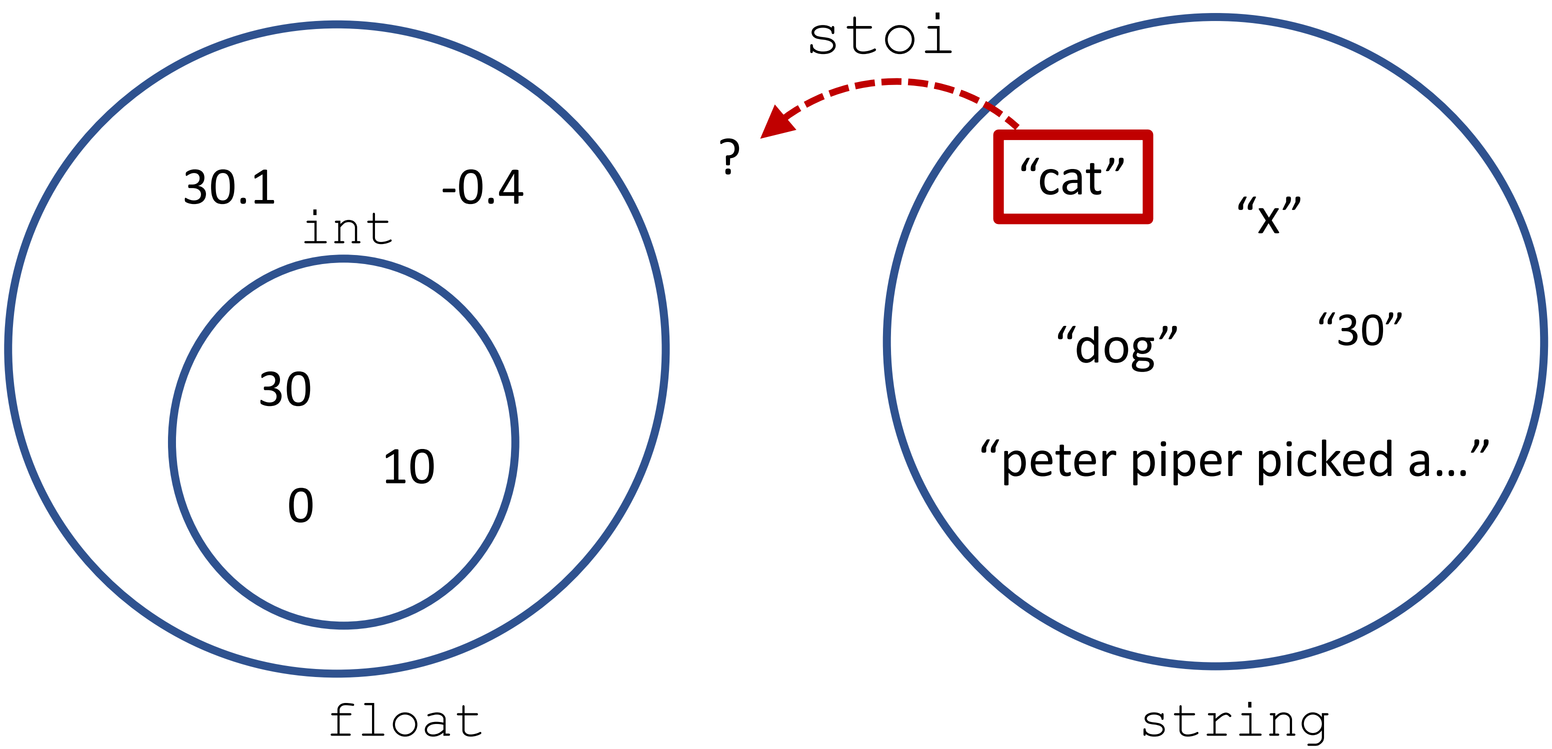


Functions : maps between types



stoi = **s**tring **to** **i**nt

Functions : maps between types



Consider exp(x)

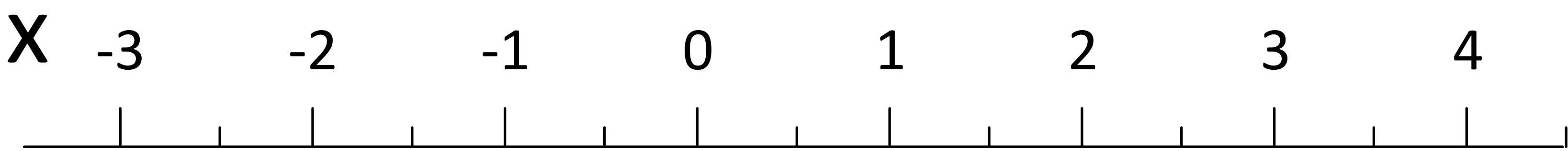
Important questions:

1. What is the input type?
2. What is the output type?
3. What is the definition?

exp(x)

$$y = \exp(x)$$

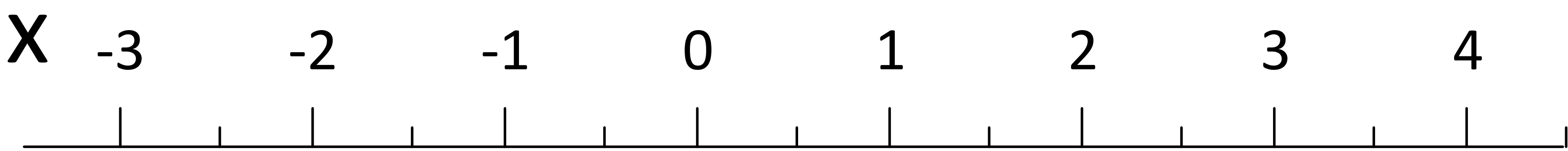
exp(x) : input range is the reals



$$x \in \mathbb{R}$$

$$y = \exp(x)$$

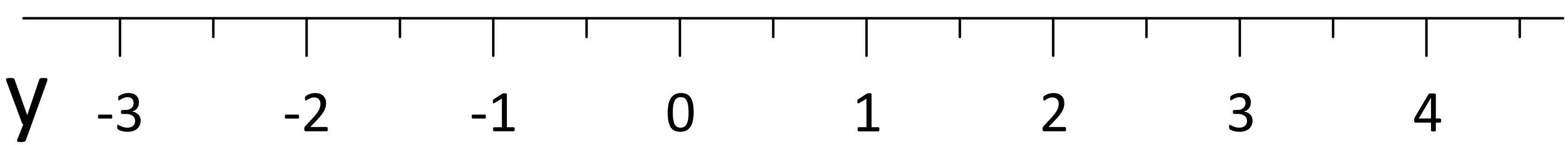
exp(x) : output range is the reals



$$x \in \mathbb{R}$$

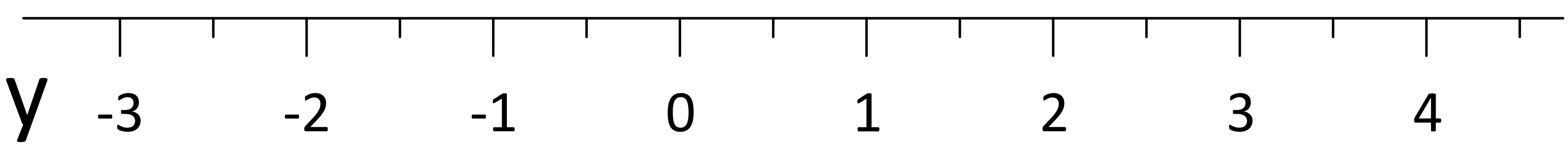
$$y = \exp(x)$$

$$y \in \mathbb{R}$$

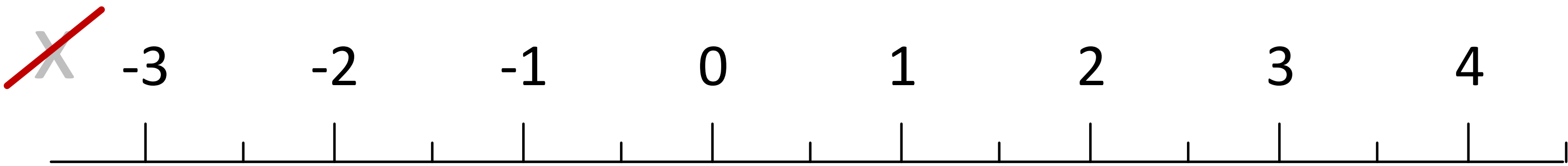


exp(x) : maps reals to reals

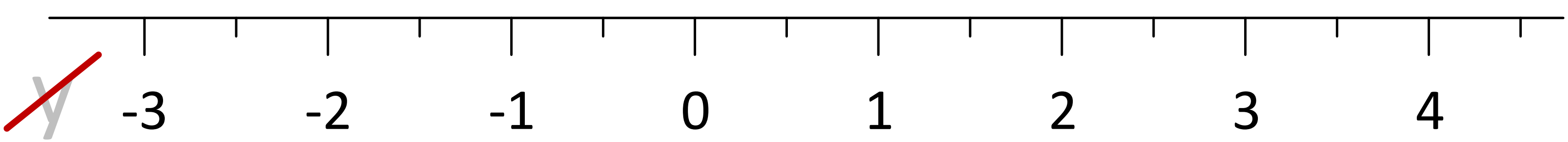
$$\exp : \mathbb{R} \rightarrow \mathbb{R}$$



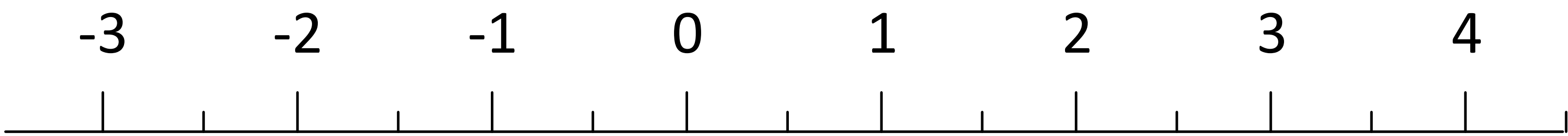
exp(x) : maps reals to reals



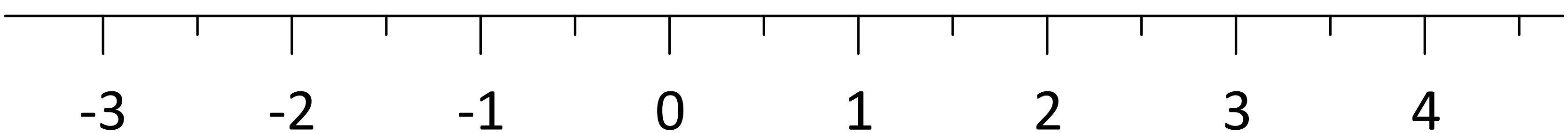
$$\exp : \mathbb{R} \to \mathbb{R}$$



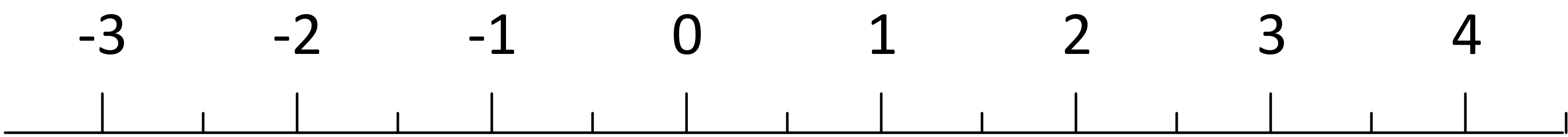
exp : maps reals to reals



$$\exp : \mathbb{R} \to \mathbb{R}$$



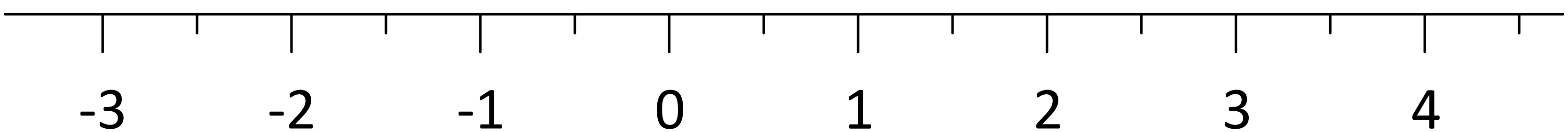
exp : maps reals to reals



$$\sin : \mathbb{R} \to \mathbb{R}$$

$$\exp : \mathbb{R} \to \mathbb{R}$$

$$\log : \mathbb{R} \to \mathbb{R}$$



exp(x) : definition using maths

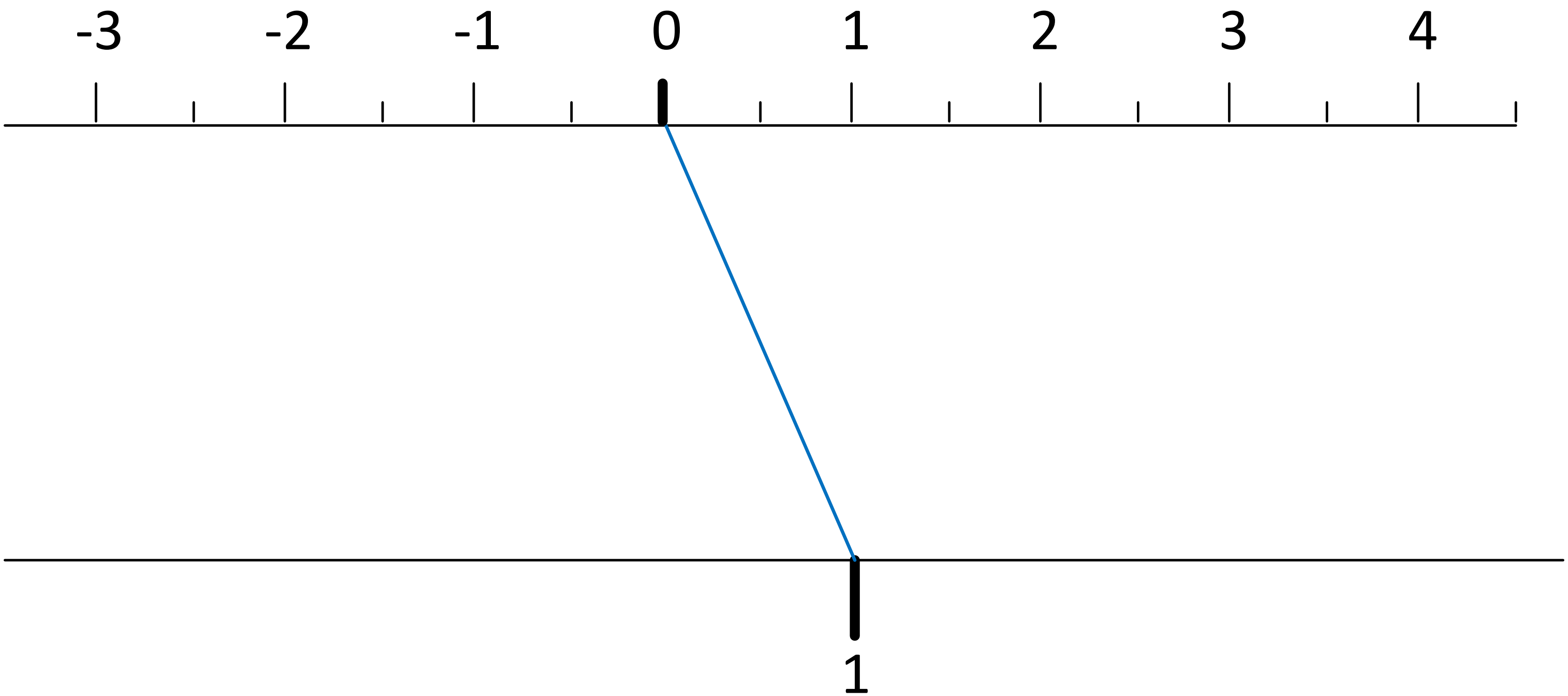
$$\exp(x) = e^x$$

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

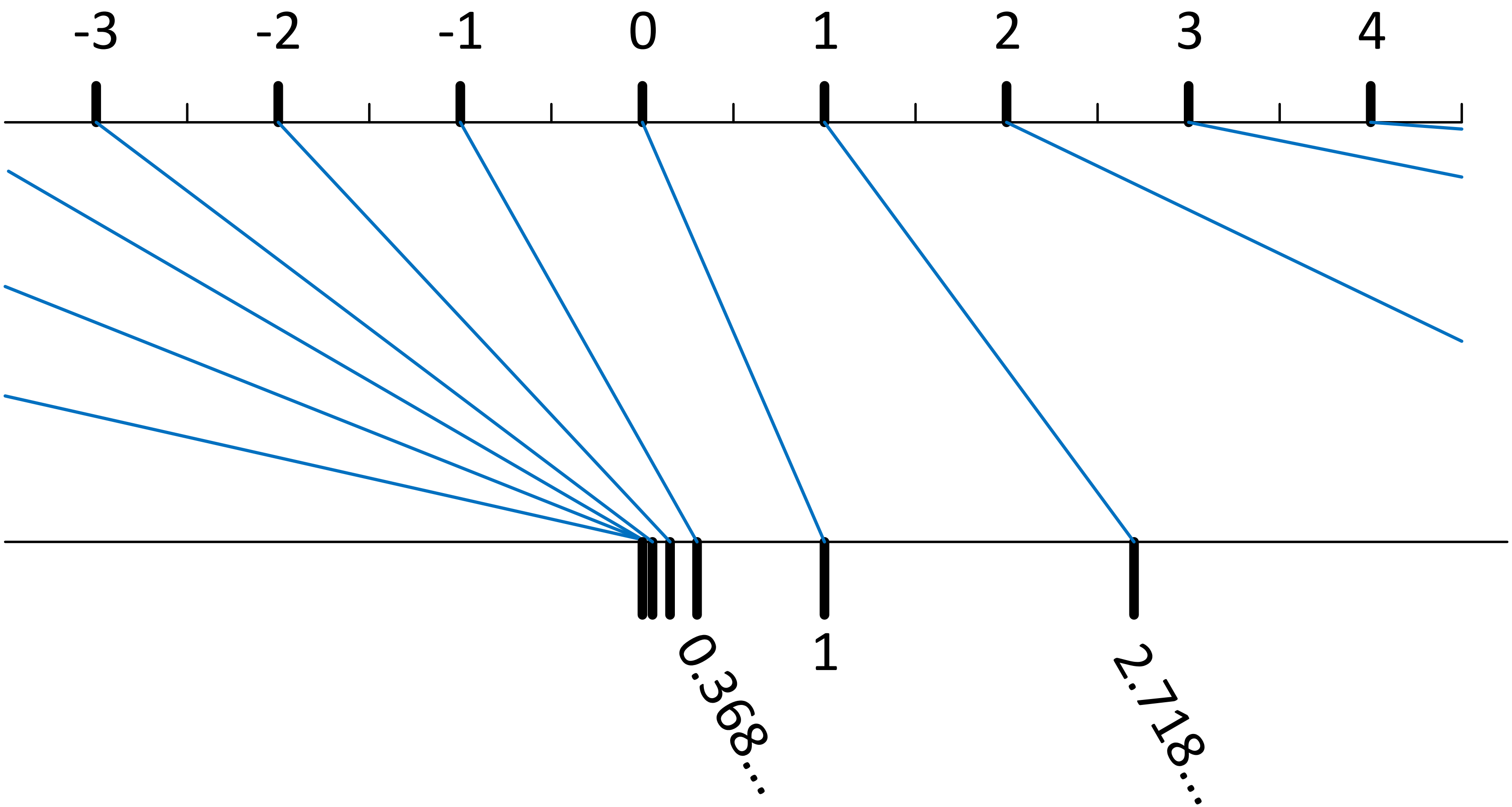
$$\exp(x) = \lim_{i \to \infty} \left(1 + {}^{x}\!/_{i}\right)^i$$

$$\exp(x) = 1 + \frac{2\tanh(x/2)}{1 - \tanh(x/2)}$$

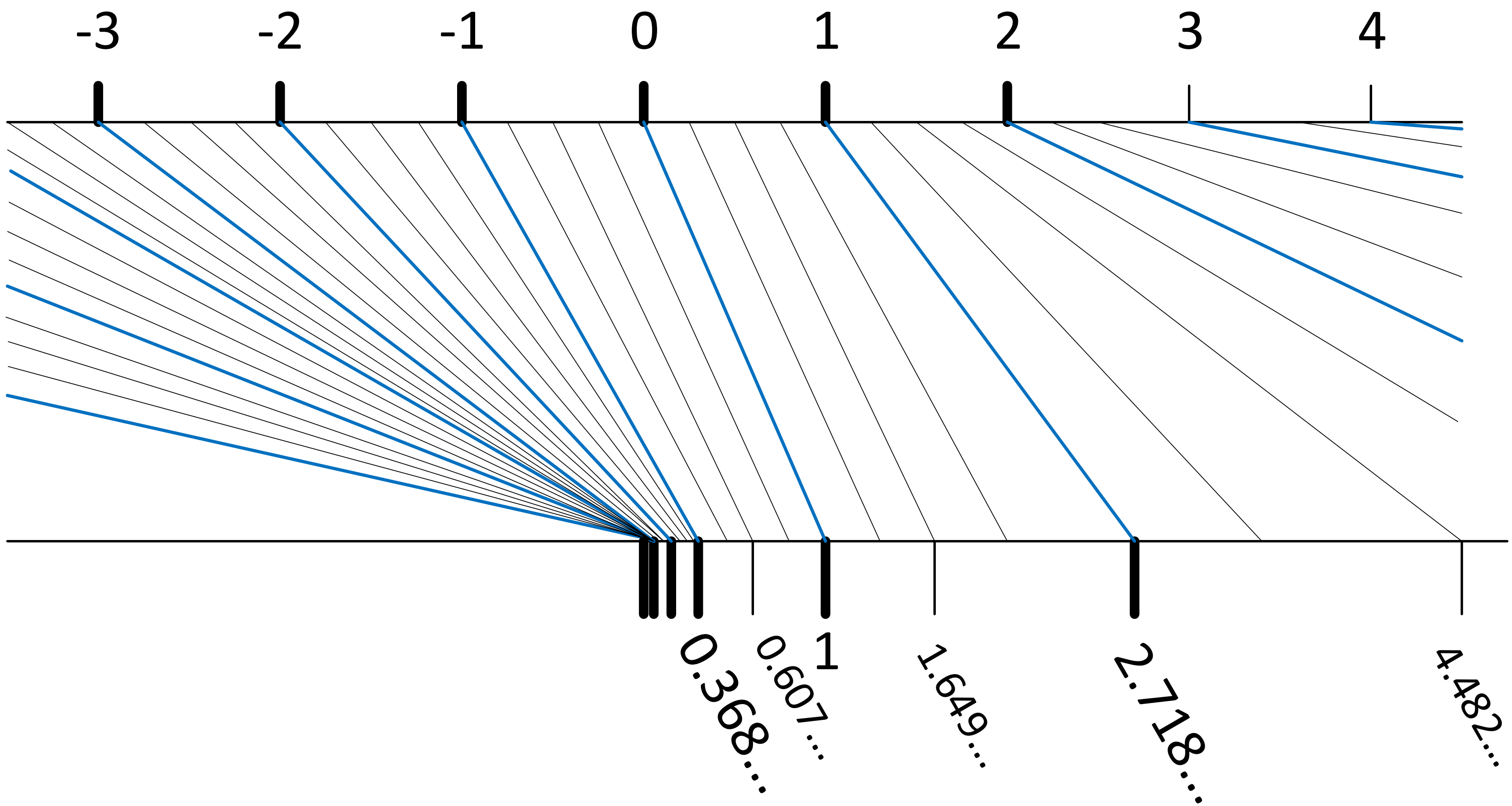
exp : definition using values



exp : definition using values



exp : definition using values



exp : definition using values

exp(-2) = 0.135335283236612691893999494972…

exp(-1) = 0.367879441171442321595523770161…

exp(0) = 1

exp(1) = 2.718281828459045235360287471352...

exp(2) = 7.389056098930650227230427460575…

Thought experiment : *Can you give any non-zero decimal number x such that* exp(*x*) *is an integer?*

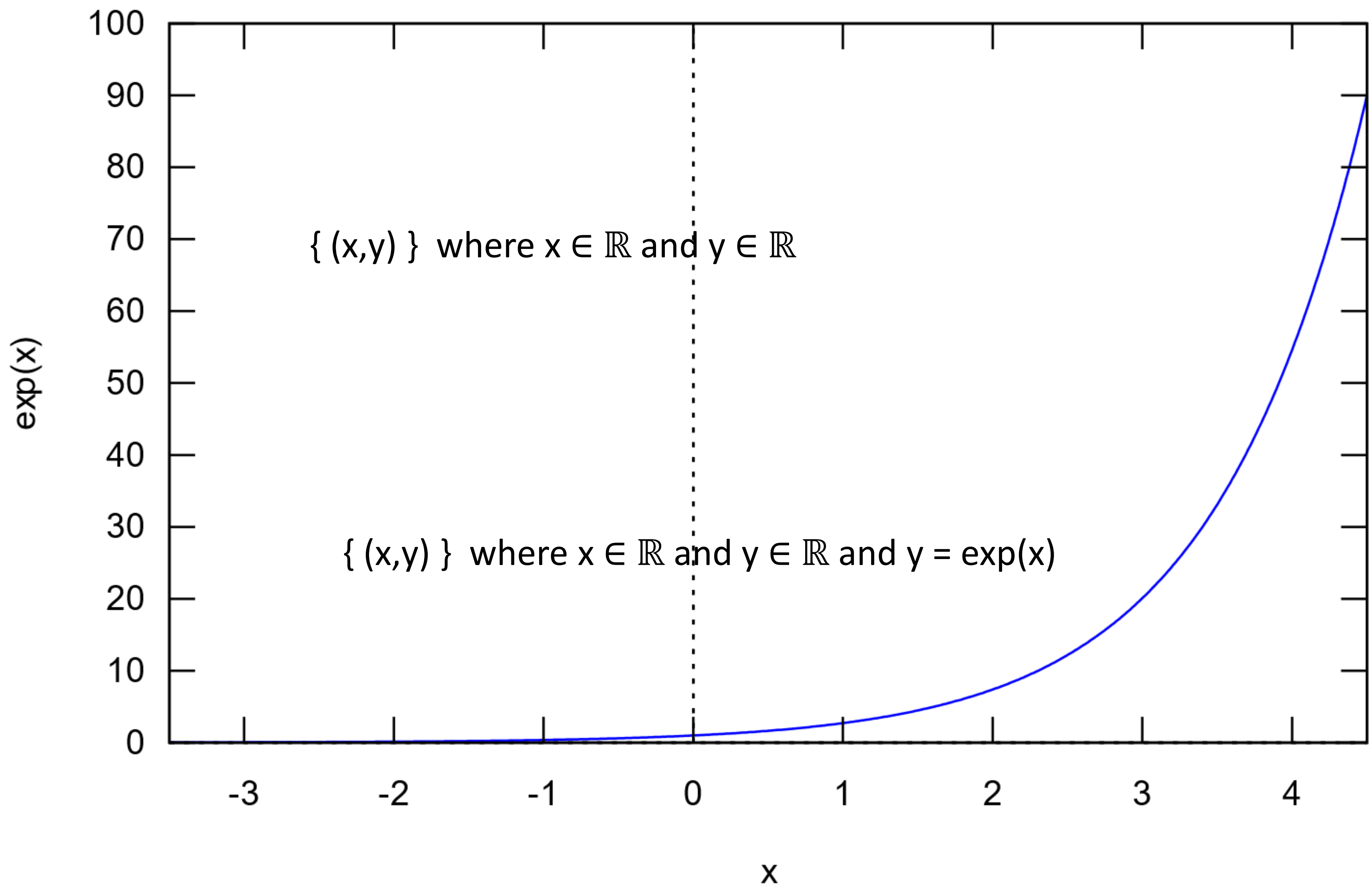
exp : definition using values

exp(-2) = 0.1353352832366126918939994949 72…
exp(-1) = 0.36787944117144232159552377016 1…
exp(0) = 1
exp(1) = 2.71828182845904523536028747135 2...
exp(2) = 7.38905609893065022723042746057 5…

The values of exp(x) are unknown and unknowable

- There is only one single “known” value: exp(0)
- All other exact values are only known implicitly
- But we can approximate to arbitrary finite accuracy

exp : definition using a graph



Mathematical functions

- Functions are mappings from sets to sets
- Many functions are exactly equivalent
- Most functions cannot be evaluated
  - An infinite number of possible input values
  - Any exact input results in an irrational number
- Mathematics specifies functions as relationships
  - Functions relate inputs to outputs

Programs must produce values

- Maths is usually not constructive
  - It tells us some value or object exists, e.g. exp(3)
  - It doesn’t tell us how to construct that value

- Programs must be constructive
  - We must construct values (answers) in finite time
  - All loops must eventually finish
  - All values must be of finite size

```cpp
#include <iostream>

using namespace std;

int main()
{
    float scale;
    cin >> scale;

    float x = 1.0;
    int i = 0;
    while( 0 < x ){
        x = x * scale;
        i = i + 1;
        cout << x << " " << i << endl;
    }
}
```

```cpp
#include <iostream>

using namespace std;

int main()
{
    float x = 1.0;
    float xp = 0.0;

    while( xp != x ){
        xp = x;
        x = x + 0.1;
    }

    cout << x << endl;
}
```

```cpp
#include <iostream>

using namespace std;

int main()
{
    int x = 5;

    while(x>0){
        x = x + 1;
    }
    cout << x << endl;
}
```

Functions in C++

Function declarations

Function declarations specify the function prototype

$$\exp : \mathbb{R} \rightarrow \mathbb{R}$$

```
float exp( float x );
```

Function declarations

Function declarations specify the function prototype

Parameter type

Function name

Parameter name

```
float exp( float x );
```

Return type

Input parameter list

Function name and parameter name are identifiers

Function declarations

Function declarations specify the function prototype

$$\exp : \mathbb{R} \to \mathbb{R}$$

```
float exp( float x );
```

Declarations are not *required* to name parameters
Just like in maths, it is the types that matter

Function declarations

Function declarations specify the function prototype

$$\exp : \mathbb{R} \to \mathbb{R}$$

```
float exp( float x );
```

Declarations are not *required* to name parameters
Just like in maths, it is the types that matter
However, it often helps users to have names

Function definitions

Function definitions specify how the function works

The function body is a list of statements

Prototype

```
float exp( float x )
{
    // Statements
}
```

Function body

Function definitions

Function definitions specify how the function works

The function body is a list of statements

Parameters can be used like variables in the body

```
float exp( float x )
{
    return 1 + x*x / 2;
}
```

$\exp(x) = 1 + x^2/2 + \ldots$

Function documentation

- The prototype says nothing about behavior
  - Only the function name and input/output types
- Behaviour needs to be observed or specified
  - What is the *function* that the function performs
- You can try to extract behavior by observation
  - What does the function do if I put in this value?
  - This does not scale – how many float inputs are there?
- Function behavior should be *specified*
  - A human-level description saying what it calculates
  - Often this is written using formal mathematics
  - This appears in the function documentation

std::exp, std::expf, std::expl

Defined in header `<cmath>`

`float exp ( float arg );` `float expf( float arg );`	(1)	(since C++11)
`double exp ( double arg );`	(2)	
`long double exp ( long double arg );` `long double expl( long double arg );`	(3)	(since C++11)
`double exp ( IntegralType arg );`	(4)	(since C++11)

Computes *e* (Euler's number, `2.7182818...`) raised to the given power `arg`

4) A set of overloads or a function template accepting an argument of any integral type. Equiva argument is cast to `double`).

Parameters

arg - value of floating-point or Integral type

Return value

If no errors occur, the base-*e* exponential of `arg` (e^{arg}) is returned.

If a range error due to overflow occurs, +HUGE_VAL, +HUGE_VALF, or +HUGE_VALL is returned.

If a range error occurs due to underflow, the correct result (after rounding) is returned.

Dilogarithm

The dilogarithm is defined as

$$Li_2(z) = -\int_0^z ds \frac{\log(1-s)}{s}$$

The functions described in this section are declared in the header file `gsl_sf_dilog.h`.

Real Argument

double **`gsl_sf_dilog`**(double *x*)

int **`gsl_sf_dilog_e`**(double *x*, gsl_sf_result * *result*)

These routines compute the dilogarithm for a real argument. In Lewin's notation this is $Li_2(x)$, the real part of the dilogarithm of a real x. It is defined by the integral representation

$$Li_2(x) = -\Re \int_0^x ds \log(1-s)/s$$

Note that $\Im(Li_2(x)) = 0$ for $x \le 1$, and $-\pi \log(x)$ for $x > 1$.

erode

Erodes an image by using a specific structuring element.

C++: void **erode**(InputArray **src**, OutputArray **dst**, InputArray **kernel**, Point **anchor**=Point(-1,-1), int **iterations**=1, int **borderType**=BORDER_CONSTANT, const Scalar& **borderValue**=morphologyDefaultBorderValue())

Python: `cv2.erode`(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]) → dst

C: void **cvErode**(const CvArr* **src**, CvArr* **dst**, IplConvKernel* **element**=NULL, int **iterations**=1)

Python: `cv.Erode`(src, dst, element=None, iterations=1) → None

Parameters:

- **src** – input image; the number of channels can be arbitrary, but the depth should be one of `CV_8U`, `CV_16U`, `CV_16S`, `` CV_32F` or ``CV_64F ``.
- **dst** – output image of the same size and type as `src`.
- **element** – structuring element used for erosion; if `element=Mat()` , a `3 x 3` rectangular structuring element is used.
- **anchor** – position of the anchor within the element; default value `(-1, -1)` means that the anchor is at the element center.
- **iterations** – number of times erosion is applied.
- **borderType** – pixel extrapolation method (see `borderInterpolate()` for details).
- **borderValue** – border value in case of a constant border (see `createMorphologyFilter()` for details).

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\texttt{dst}(x,y) = \min_{(x',y'):\,\texttt{element}(x',y') \ne 0} \texttt{src}(x+x',y+y')$$

Function calls

A function call evaluates the function

Function arguments are run-time input values

```
float exp( float x );

int main()
{
    float y = exp( 0.2 );
}
```

Argument

Function call

Function return value

Functions return a value using the return statement

Return type

Return value

```
float exp( float x )
{
    return 1 + x*x/2 + x*x*x/6;
}
```

Return statement

Parameter lists

You can have zero or more parameters

Each parameter can be a different type

```
int add(int a, float b);

string choose(bool b, string x, string y);

int sum(vector<int> inputs);
```

There are performance implications for choice of parameter types;
for now we ***only*** care about functional correctness – does it work?

Function bodies : do what you like

```
string choose(bool b, string x, string y)
{
    if(b){
        return x;
    }else{
        return y;
    }
}

int sum(vector<int> inputs)
{
    int res;
    for(int i=0; i< inputs.size(); i=i+1){
        res = res + inputs[i];
    }
    return res;
}
```

Declarations from `#include`

`#include` is like copying and pasting code

`<cmath>` is the file `/usr/include/c++/7/cmath`

```
#include <cmath>

int main()
{
    float y = exp( 0.2 );
}
```

```
    typename __gnu_cxx::__enable_if<__is_integer<_Tp>::__value,
                                    double>::__type
    cosh(_Tp __x)
    { return __builtin_cosh(__x); }

  using ::exp;

#ifndef __CORRECT_ISO_CPP_MATH_H_PROTO
  inline _GLIBCXX_CONSTEXPR float
  exp(float __x)
  { return __builtin_expf(__x); }

  inline _GLIBCXX_CONSTEXPR long double
  exp(long double __x)
  { return __builtin_expl(__x); }
#endif

  template<typename _Tp>
    inline _GLIBCXX_CONSTEXPR
    typename __gnu_cxx::__enable_if<__is_integer<_Tp>::__value,
                                    double>::__type
    exp(_Tp __x)
    { return __builtin_exp(__x); }

  using ::fabs;
```

```
    typename __gnu_cxx::__enable_if<__is_integer<_Tp>::__value,
                                    double>::__type
    cosh(_Tp __x)
    { return __builtin_cosh(__x); }

  using ::exp;

#ifndef __CORRECT_ISO_CPP_MATH_H_PROTO
  inline _GLIBCXX_CONSTEXPR float
  exp(float __x)
  { return __builtin_expf(__x); }

  inline _GLIBCXX_CONSTEXPR long double
  exp(long double __x)
  { return __builtin_expl(__x); }
#endif

  template<typename _Tp>
    inline _GLIBCXX_CONSTEXPR
    typename __gnu_cxx::__enable_if<__is_integer<_Tp>::__value,
                                    double>::__type
    exp(_Tp __x)
    { return __builtin_exp(__x); }

  using ::fabs;
```

```
    typename __gnu_cxx::__enable_if<__is_integer<_Tp>::__value,
                                    double>::__type
    cosh(_Tp __x)
    { return __builtin_cosh(__x); }

  using ::exp;

#ifndef __CORRECT_ISO_CPP_MATH_H_PROTO
  inline _GLIBCXX_CONSTEXPR
  float exp(float __x)
  { return __builtin_expf(__x); }

  inline _GLIBCXX_CONSTEXPR long double
  exp(long double __x)
  { return __builtin_expl(__x); }
#endif

  template<typename _Tp>
    inline _GLIBCXX_CONSTEXPR
    typename __gnu_cxx::__enable_if<__is_integer<_Tp>::__value,
                                    double>::__type
    exp(_Tp __x)
    { return __builtin_exp(__x); }

  using ::fabs;
```

Declarations from `#include`

`#include` is like copying and pasting code

`<cmath>` is the file `/usr/include/c++/7/cmath`

```
// Begin from <cmath>
float exp(float __x);
// End from cmath

int main()
{
    float y = exp( 0.2 );
}
```

Declarations versus definitions

- Declarations introduce function prototypes
  - You cannot call a function until it is declared
  - `#include` is mainly declaring functions (and types)

- You can declare and define at the same time
  - Often the most convenient methods
- Separate declaration and definition has advantages
  - System calls : where is the definition of `log` from `<cmath>`?
  - Recursive functions : next week

Functions as abstractions

Functions allow us to break up programs

- *Prototype*: what are the input and output types?
- *Specification*: what is it supposed to calculate?
- *Implementation*: how does it calculate it?

```
/* Calculates exponential
   of the input value */
float exp(float x);
```

$$\exp = x \mapsto e^x$$

$$\exp : \mathbb{R} \to \mathbb{R}$$

Reasoning about functions

- Think of substitutions

```
int twice(int x)
{
    return x * 2;
}

int main()
{
    int a = 1;
    int b = twice( a );
}
```

Reasoning about functions

- Think of substitutions

```
int twice(int x)
{
    return x * 2;
}

int main()
{
    int a = 1;
    int b = twice( a );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values

```
int twice(int x)
{
    return x * 2;
}

int main()
{
    int a = 1;
    int b = twice( a );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values

```
int twice(int x = 1)
{
    return x * 2;
}

int main()
{
    int a = 1;
    int b = twice( 1 );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values

```
int twice(int x = 1)
{
    return 1 * 2;
}

int main()
{
    int a = 1;
    int b = twice( 1 );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values

```
int twice(int x = 1)
{
    return 2;
}

int main()
{
    int a = 1;
    int b = twice( 1 );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values
  - Note: conceptual only!
  - Evaluation is at run-time

```
int twice(int x = 1)
{
    return 2;
}

int main()
{
    int a = 1;
    int b = 2 ;
}
```

Reasoning about functions

- Think of substitutions
- Substitute values
  - Note: conceptual only!
  - Evaluation is at run-time

```
int twice(int x)
{
    return x * 2;
}

int main()
{
    int a = 1;
    int b = twice( a );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values
  - Note: conceptual only!
  - Evaluation is at run-time
- Substitute expressions

```
int twice(int x)
{
    return x * 2;
}

int main()
{
    int a = 1;
    int b = twice( a );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values
  - Note: conceptual only!
  - Evaluation is at run-time
- Substitute expressions

```
int twice(int x = a)
{
    return x * 2;
}

int main()
{
    int a = 1;
    int b = twice( a );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values
  - Note: conceptual only!
  - Evaluation is at run-time
- Substitute expressions

```
int twice(int x = a)
{
    return a * 2;
}

int main()
{
    int a = 1;
    int b = twice( a );
}
```

Reasoning about functions

- Think of substitutions
- Substitute values
  - Note: conceptual only!
  - Evaluation is at run-time
- Substitute expressions
  - Again: conceptual only!
  - No variable `a` in `twice`

```
int twice(int x = a)
{
    return a * 2;
}

int main()
{
    int a = 1;
    int b =  a * 2;
}
```

Next time : functions + structs

- Functions and variable lifetimes
  - When do variables in functions get created?

- Structs : heterogenous types
  - Passing complex data types in and out of functions