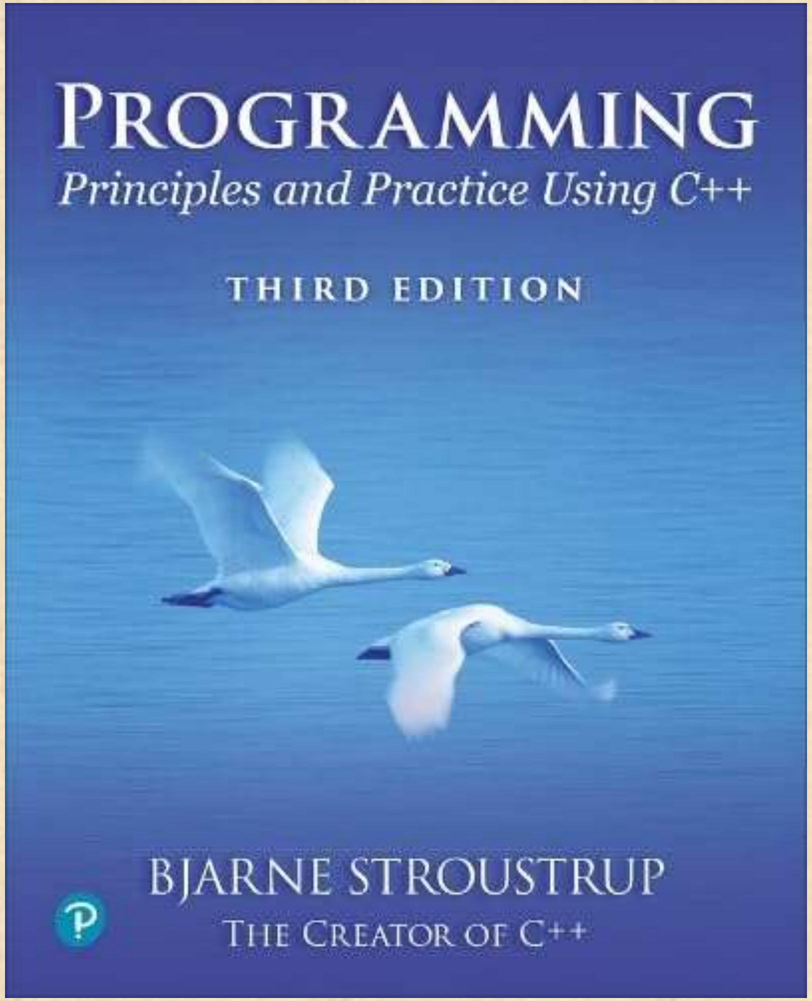
Chapter 13 – Graphing Functions and Data



The best is the enemy of the good.
– Voltaire

Abstract

- Here we present ways of graphing functions and data and some of the programming techniques needed to do so, notably scaling.
  - Graphical function objects
  - Approximation and precision
  - Scaling and data
  - Layout

Note

- This course is about programming
  - The examples – such as graphics – are simply examples of
    - Useful programming techniques
    - Useful tools for constructing real programs

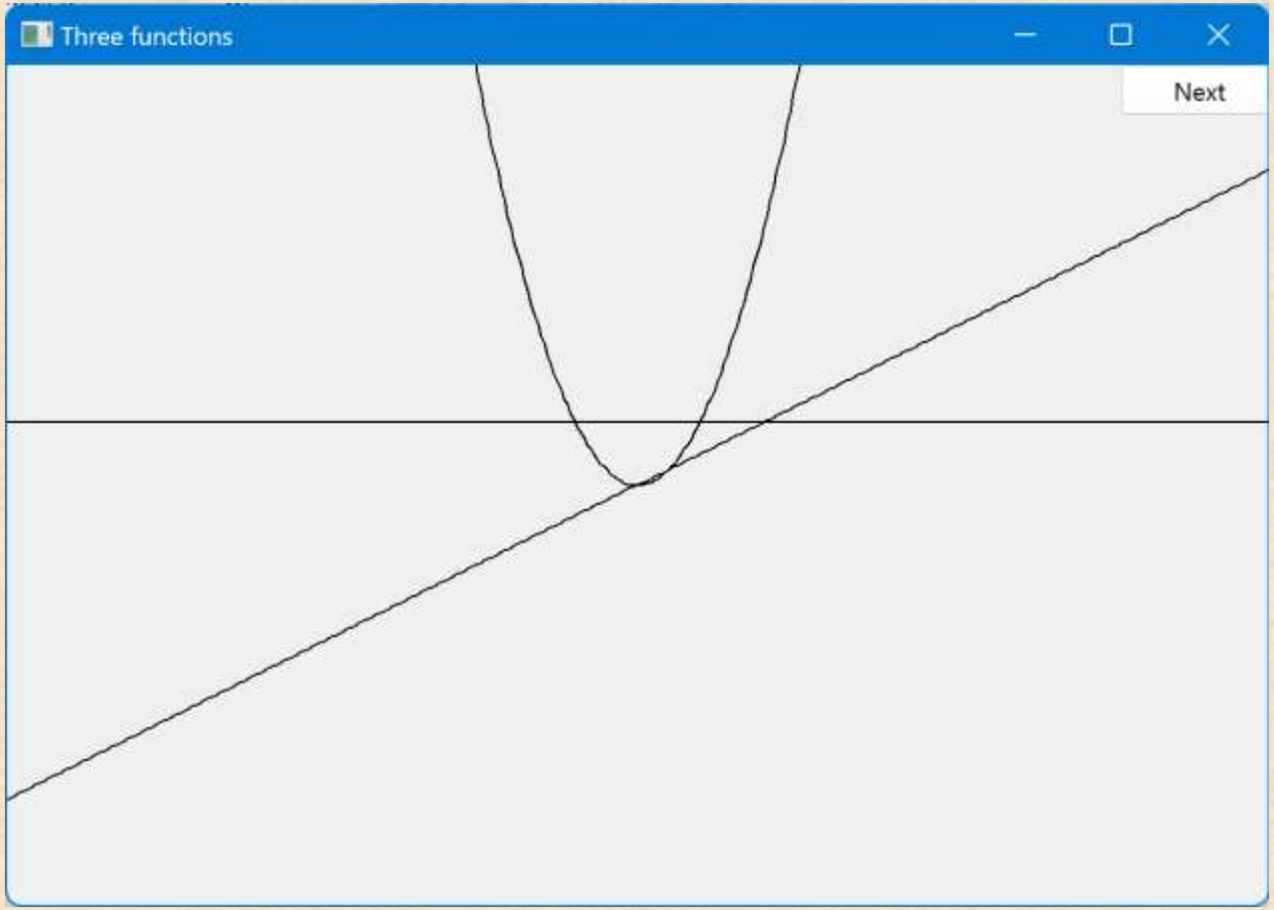
    Look for the way the examples are constructed
  - How are “big problems” broken down into little ones and solved separately?
  - How are classes defined and used?
    - Do they have sensible data members?
    - Do they have useful member functions?
  - Use of variables
    - Are there too few?
    - Too many?
    - How would you have named them better?

Graphing functions

- For any new tool, technique, library, or language
  - Start with something really simple
  - Always remember "Hello, World!"
- We graph functions of one argument yielding one value
  - Plot (x, f(x)) for values of x in some range [r1,r2)
- Let's graph three simple functions:

```
double one(double x) { return 1; }              // y==1
double slope(double x) { return 0.5*x; }        // slope is 0.5
double square(double x) { return x*x; }         // y==x*x
```

Functions




```
double one(double x) { return 1; }               // y==1
double slope(double x) { return 0.5*x; }         // y==0.5*x
double square(double x) { return x*x; }          // y==x*x
```

We need some Constants to control presentation

- *Choosing a center (0,0), scales, and number of points can be fiddly*
- *The range usually comes from the definition of what you are doing*

```
const int xmax = 600;                    // window size (600 by 400)
const int ymax = 400;

const int x_orig = xmax/2;
const int y_orig = ymax/2;
const Point orig {x_orig, y_orig};       // position of Cartesian (0,0) in window

const int r_min = -10;                   // range [-10:11) == [-10:10] of x
const int r_max = 11;

const int n_points = 400;                // number of points used in range

const int x_scale = 30;                  // scaling  factors
const int y_scale = 30;
```

How do we write code to do this?

```
Simple_window win {Point{100,100},xmax,ymax,"Three function"};

Function s   {one,          -10,11,   orig,   n_points, x_scale,y_scale};
Function s2 {slope,      -10,11,   orig,   n_points, x_scale,y_scale};
Function s3 {(square, -10,11,   orig,   n_points, x_scale,y_scale};

win.attach(s);
win.attach(s2);
win.attach(s3);
win.wait_for_button();
```

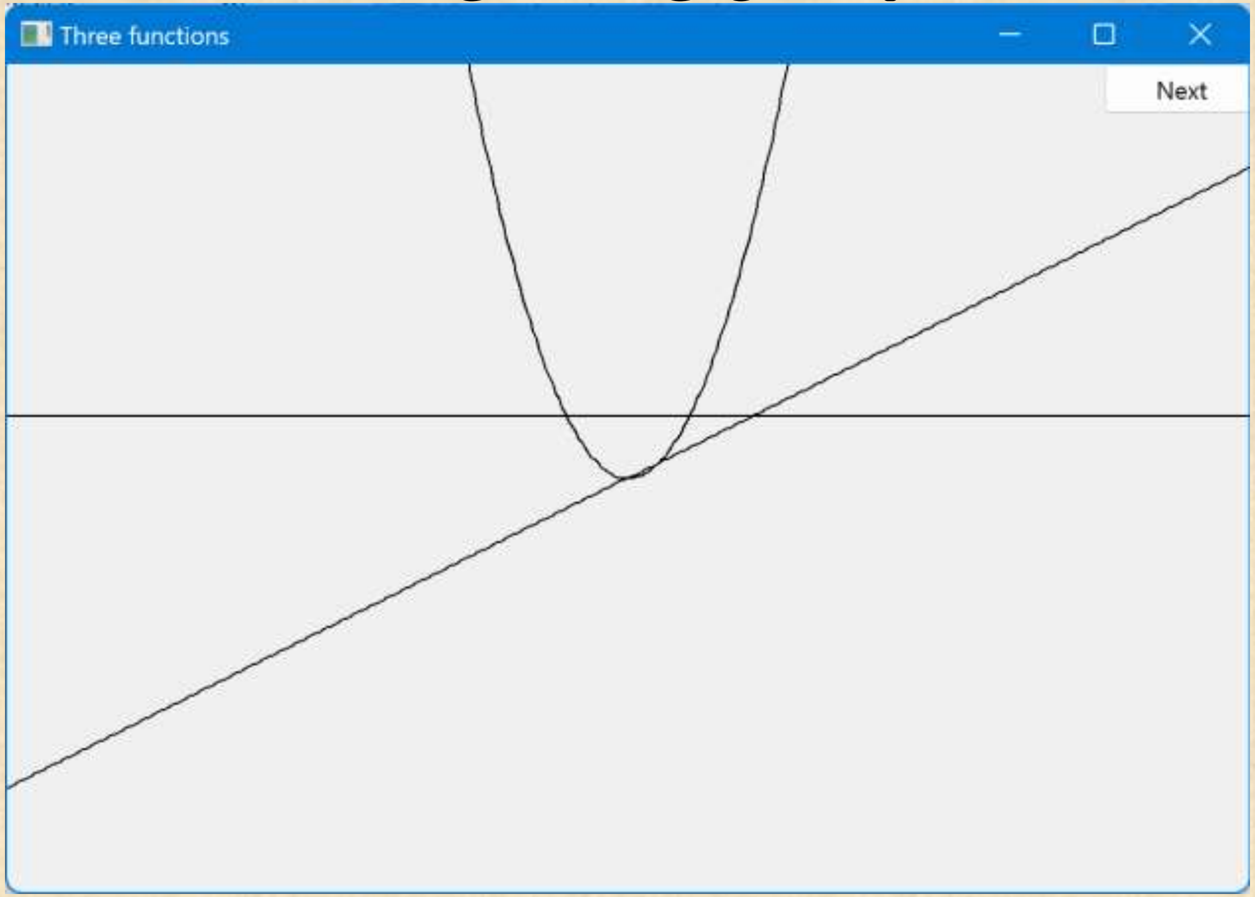
Function to be graphed

First point

Range in which to graph [x0:xN)

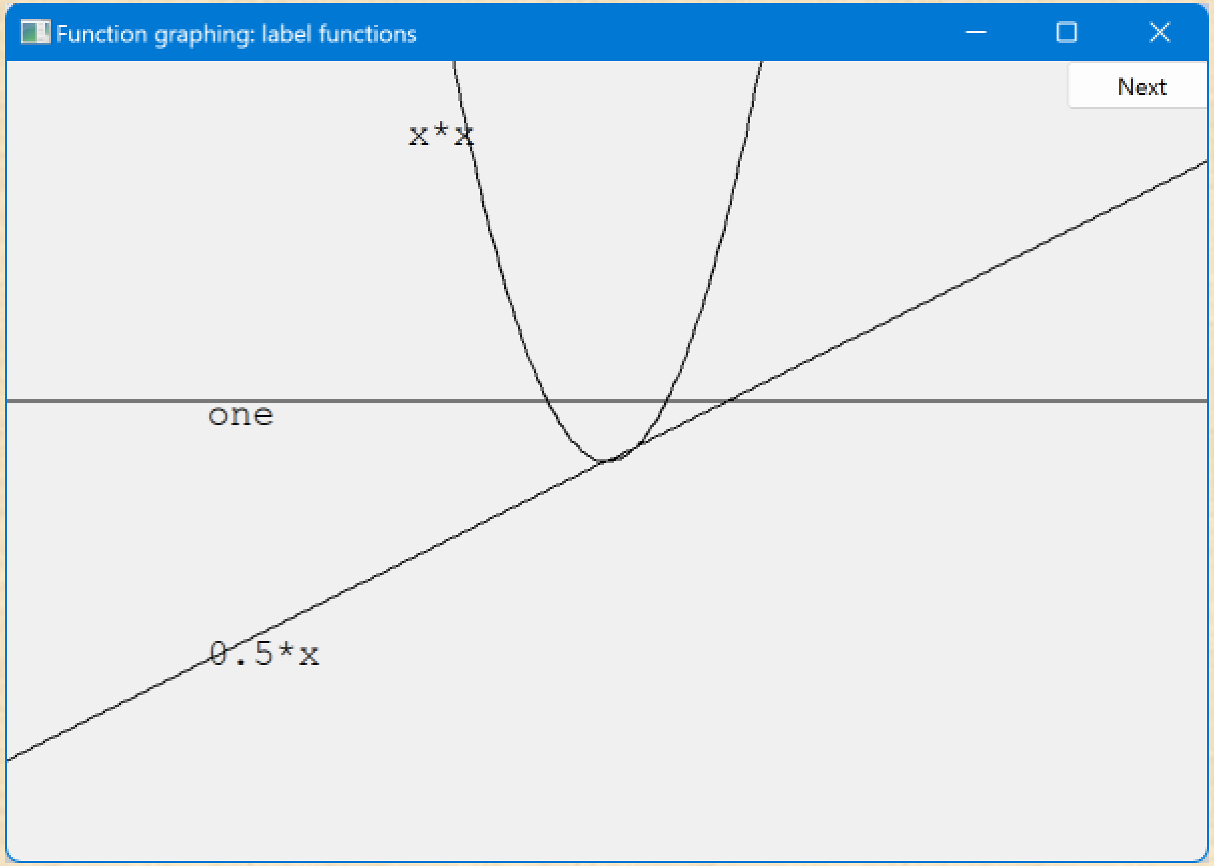
“stuff” to make the graph fit into the window

Functions – but what does it mean?



- What's wrong with this?
  - No axes (no scale)
  - No labels

Label the functions

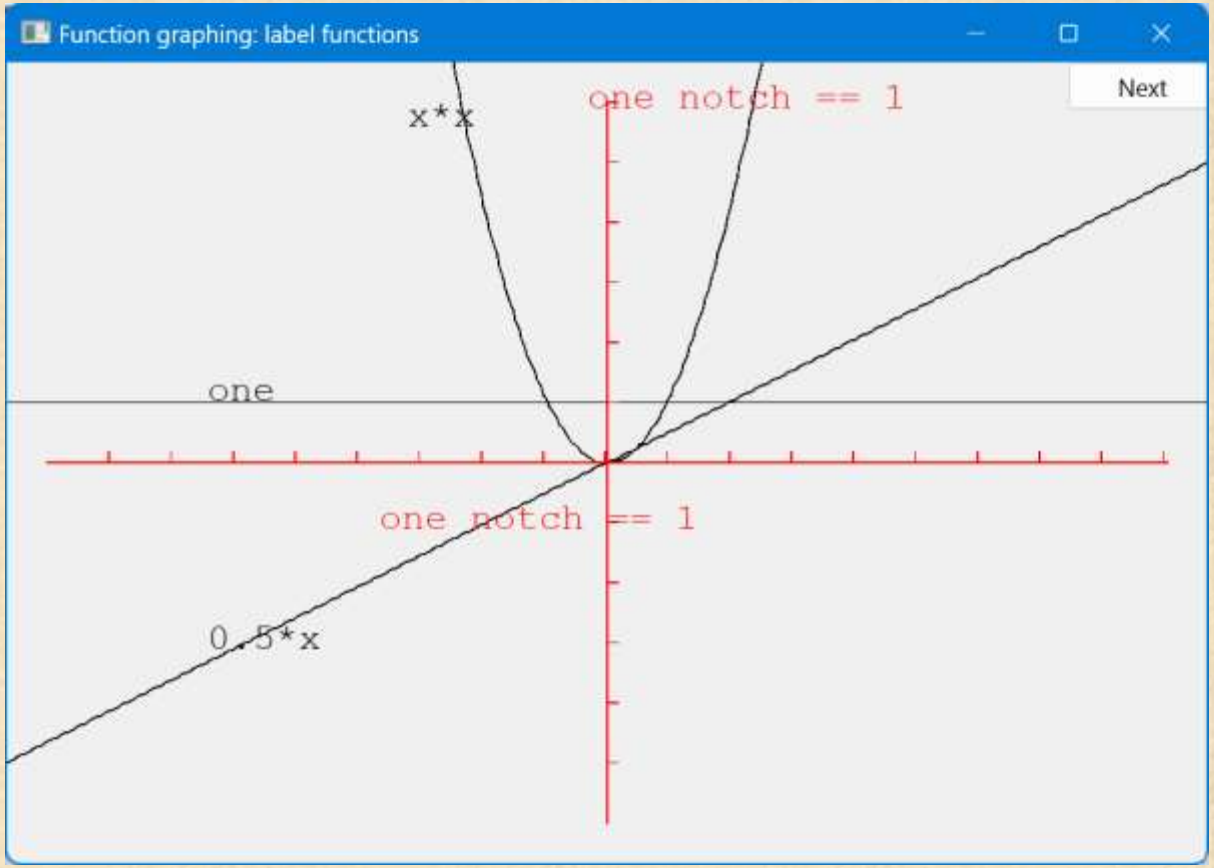


Text ts {Point{100,y_orig-40},"one"};

Text ts2 {Point{100,y_orig+y_orig/2-20},"0.5*x};

Text ts3 {Point{x_orig-90,20),"x*x"};

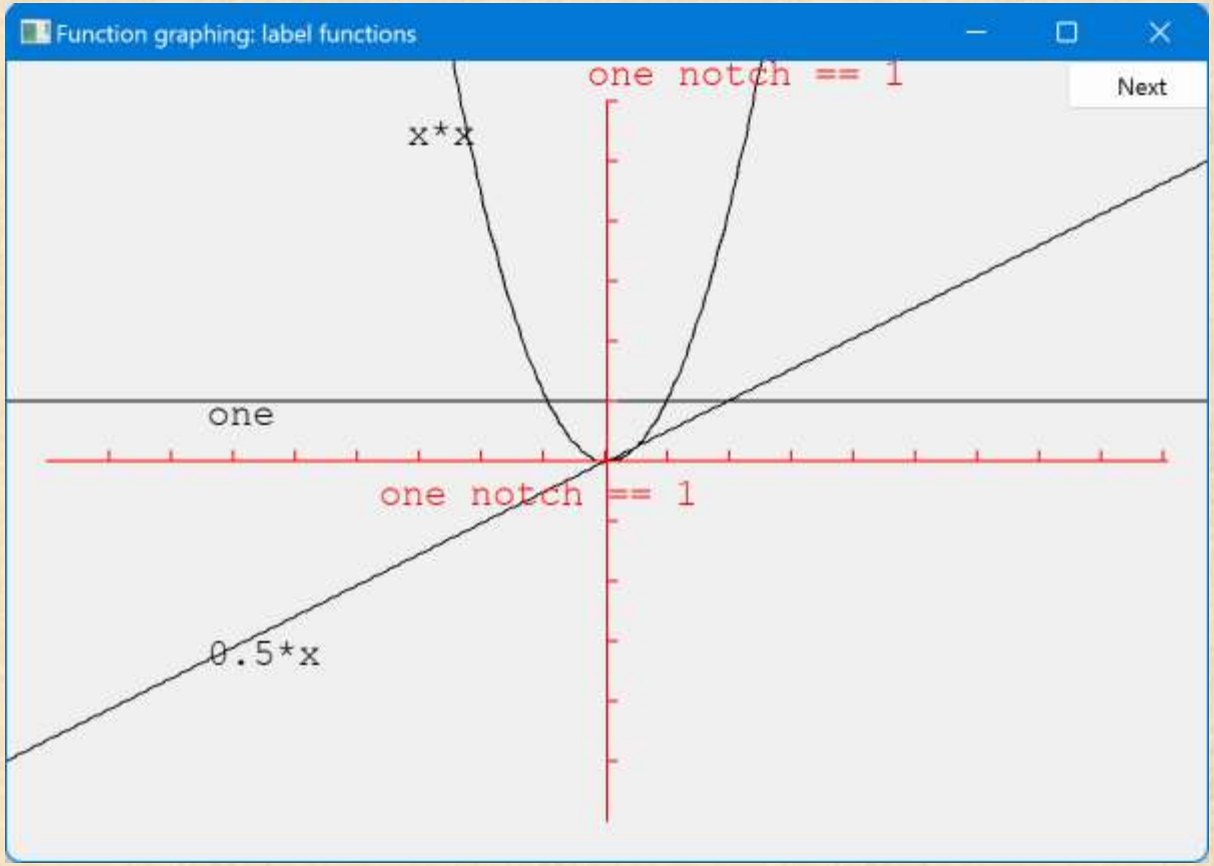
Add x-axis and y-axis



- We can use axes to show (0,0) and the scale

```
Axis x {Axis::x, Point{20,y_orig},  xlength/x_scale, "one notch == 1"};
Axis y {Axis::y, Point{x_orig, ylength+20}, ylength/y_scale, "one notch == 1"};
```

Use color (in moderation)




```
x.set_color(Color::red);
y.set_color(Color::red);
```

The implementation of Function

- We need a type for the argument specifying the function to graph
  - **using** can be used to declare a new name for a type
    - **using Count = int;** *// now Count means int*

  - Define the type of our desired argument, **Fct**
    - **using Fct = std::function<(double(double)>;** *// the type of a function*
      // taking a double argument
      // and returning a double
  - Examples of functions of type **Fct**:

```
double one(double x) { return 1; }          // y==1
double slope(double x) { return 0.5*x; }    // y==0.5*x
double square(double x) { return x*x; }     // y==x*x
```

Now Define "Function"

- We store the function as a sequence of line segments in a polyline

```
struct Function : Open_polyline {                // all it needs is a constructor!
        Function(
                Fct f,
                double r1, double r2,                      // range
                Point orig,
                int count = 100,                           // Number of line segments
                double xscale = 25, double yscale = 25     // x and y scaling
        );
        // the function parameters are not stored
};
```

Implementation of Function

```
Function::Function( Fct f,
                    double r1, double r2,
                    Point xy,
                    int count,
                    double xscale, double yscale )
{
  if (r2-r1<=0) error("bad graphing range");
  if (count<=0) error("non-positive graphing count");
  double dist = (r2-r1)/count;
  double r = r1;
  for (int i = 0; i<count; ++i) {
        add(Point{xy.x+int(r*xscale), xy.y-int(f(r)*yscale)});
        r += dist;
  }
}
```

Default arguments

- Seven arguments are too many!
  - Many too many
    - We're just asking for confusion and errors
  - Provide defaults for some (trailing) arguments
    - Default arguments are often useful for constructors

```
struct Function : Open_polyline {
    Function(Fct f, double r1, double r2, Point xy,
                Count count = 100, double xscale = 25, double yscale=25 );
};

Function f1 {sqrt, 0, 11, orig, 100, 25, 25};        // ok (obviously)
Function f2 {sqrt, 0, 11, orig, 100, 25};            // ok: exactly the same as f1
Function f3 {sqrt, 0, 11, orig, 100};                // ok: exactly the same as f1
Function f4 {sqrt, 0, 11, orig};                     // ok: exactly the same as f1
```

Function

- Is **Function** a “pretty class”?
  - No
    - Why not?
  - What could you do with all those position and scaling arguments?
    - See §13.6.3 for one minor idea
  - If you can’t do something genuinely clever, do something simple, so that the user can do anything needed
    - Such as adding parameters so that the caller can control precision
  - Use default argument to simplify the calling interface

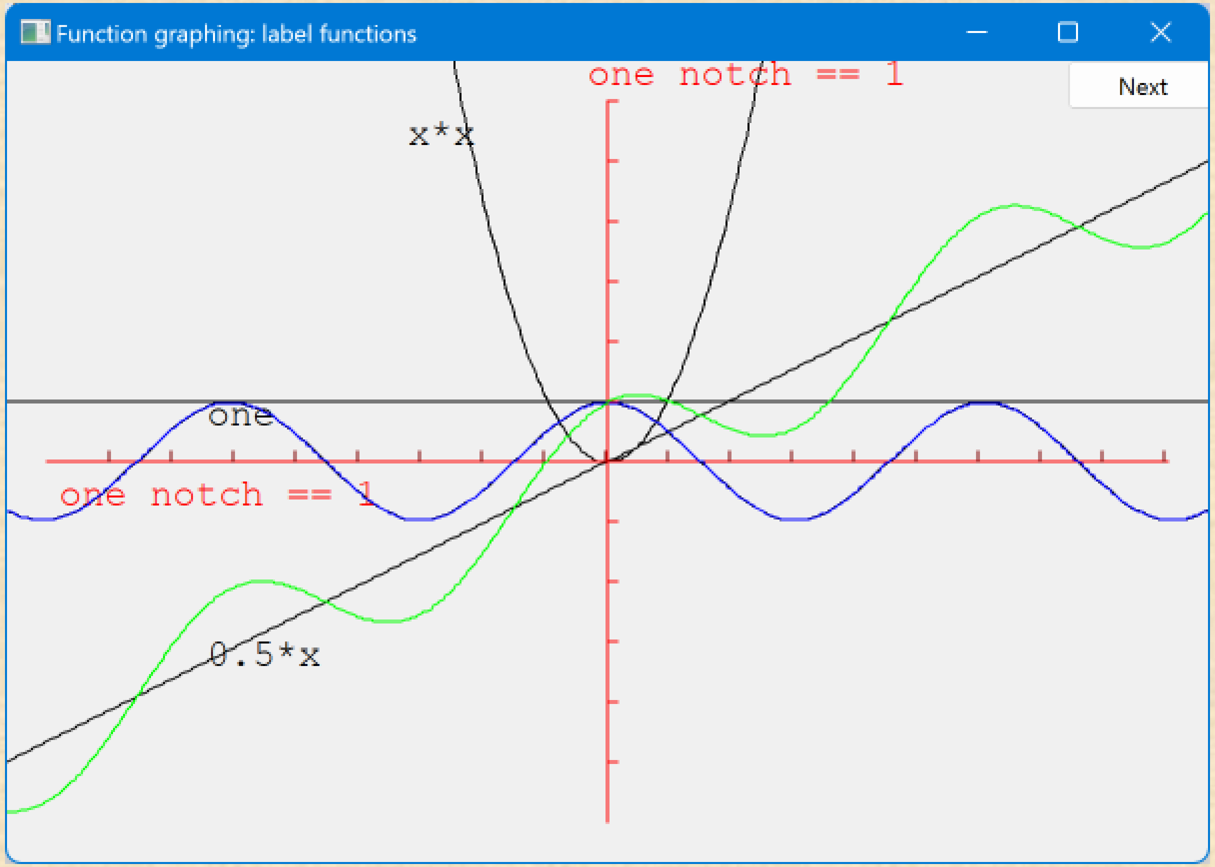
Some more functions

```
// You can combine functions (e.g., by addition):
double sloping_cos(double x) { return cos(x)+slope(x); }

// cos() is overloaded, here we must say which versions we want
double dcos(double d) { return cos(d); }   // dcos() chooses cos(double)

Function s4{ dcos,r_min,r_max,orig,400,30,30 };
s4.set_color(Color::blue);
Function s5{ sloping_cos, r_min,r_max,orig,400,30,30 };
s5.set_color(Color::green);
```

Cos and sloping-cos



Some standard mathematical functions

- **double abs(double);** *// absolute value*

- **double ceil(double d);** *// smallest integer **>= d***
- **double floor(double d);** *// largest integer **<= d***

- **double sqrt(double d);** *// **d** must be non-negative*

- **double cos(double);**
- **double sin(double);**
- **double tan(double);**
- **double acos(double);** *// result is non-negative; "a" for "arc"*
- **double asin(double);** *// result nearest to 0 returned*
- **double atan(double);**
- **double sinh(double);** *// "h" for "hyperbolic"*
- **double cosh(double);**
- **double tanh(double);**

Some standard mathematical functions

- **double exp(double);** *// base e*
- **double log(double d);** *// natural logarithm (base e) **; d** must be positive*
- **double log10(double);** *// base 10 logarithm*

- **double pow(double x, double y);** *// **x** to the power of **y***
- **double pow(double x, int y);** *// **x** to the power of **y***
- **double atan2(double y, double x);** *// atan(y/x)*
- **double fmod(double d, double m);** *// floating-point remainder; same sign as d%m*
- **double ldexp(double d, int i);** *// d*pow(2,i)*

Why graphing?

- Because you can see things in a graph that are not obvious from a set of numbers
  - How would you understand a sine curve if you couldn't (ever) see one?
- Visualization
  - Is key to understanding in many fields
  - Is used in most research and business areas
    - Science, medicine, business, telecommunications, control of large systems
  - Can communicate large amounts of data simply

An example: e^x

$e^x == 1$

$+ x$

$+ x^2/2!$

$+ x^3/3!$

$+ x^4/4!$

$+ x^5/5!$

$+ x^6/6!$

$+ x^7/7!$

$+ \ldots$

Where ! means factorial (e.g. 4!==4*3*2*1)

(This is the Taylor series expansion e^x about x==0)

Simple algorithm to approximate e^x

```
double fac(int n) { /* … */ }          // factorial, n! == n*(n-1)* … *2

double term(double x, int n)          // x^n/n!
{
    return pow(x,n)/fac(n);
}

double exp_n(double x, int n)          // sum of n terms of Taylor series for e^x
{
    double sum = 0;
    for (int i = 0; i<n; ++i)
            sum+=term(x,i);
    return sum;
}
```

"Animate" approximations to e^x

("Boilerplate")

```
Application app;

constexpr int xmax = 600;              // window size
constexpr int ymax = 400;

constexpr int x_orig = xmax / 2;       // position of (0,0) is center of window
constexpr int y_orig = ymax / 2;
constexpr Point orig{ x_orig,y_orig };

constexpr int r_min = -10;              // range [-10:11)
constexpr int r_max = 11;

constexpr int n_points = 400;          // number of points used in range

constexpr int x_scale = 30;            // scaling factors
constexpr int y_scale = 30;
```

"Animate" approximations to e^x ("Boilerplate")

```
Simple_window win{ Point{100,100},xmax,ymax,"Real exp" };
constexpr int xlength = xmax - 40;                // make the axis a bit smaller than the window
constexpr int ylength = ymax - 40;
Axis x{ Axis::x,Point{20,y_orig}, xlength, xlength / x_scale, "one notch == 1" };
Axis y{ Axis::y,Point{x_orig, ylength + 20}, ylength, ylength / y_scale, "one notch == 1" };

x.set_color(Color::red);
y.set_color(Color::red);
// what we are trying to approximate:
Function real_exp{ [](double d) { return exp(d); },r_min,r_max,orig,200,x_scale,y_scale };
real_exp.set_color(Color::blue);

win.attach(real_exp);
win.attach(x);
win.attach(y);

win.wait_for_button();
```

"Animate" approximations to e^x

```
for (int n = 0; n<50; ++n) {
    ostringstream ss;
    ss << "exp approximation; n==" << n ;
    win.set_label(ss.str().c_str());

    // next approximation:
    Function e([n](double x) { return exp_n(x,n); },        // n terms of Taylor series
               r_min,r_max,orig,200,x_scale,y_scale);

    win.attach(e);
    win.wait_for_button();        // give the user time to look
    win.detach(e);
}
```

Lambda expression

Lambda expression

- What was this?
  - **([n](double x) { return exp_n(x,n); }** *// n terms of Taylor series*
- It's a lambda, aka lambda expression aka lambda function
  - It takes **n** from the context and makes a function object using it
- we can only graph functions of one argument,
  - so we had the language write one for us (grabbing the "n" from the context)
- **[n](double x) { return expe(x,n); }**

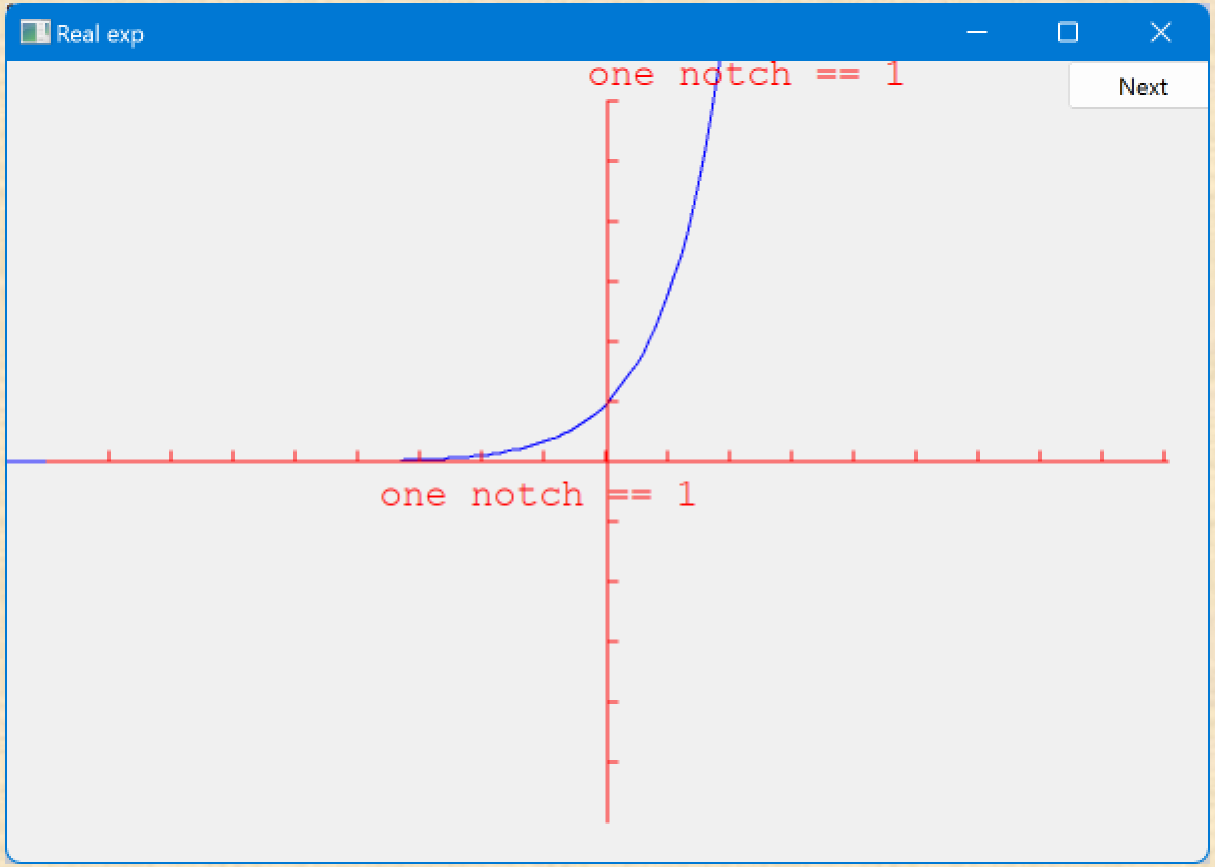
Capture list
starts with **[**

Argument declaration
starts with **(**

lambda function body
Starts with **{**

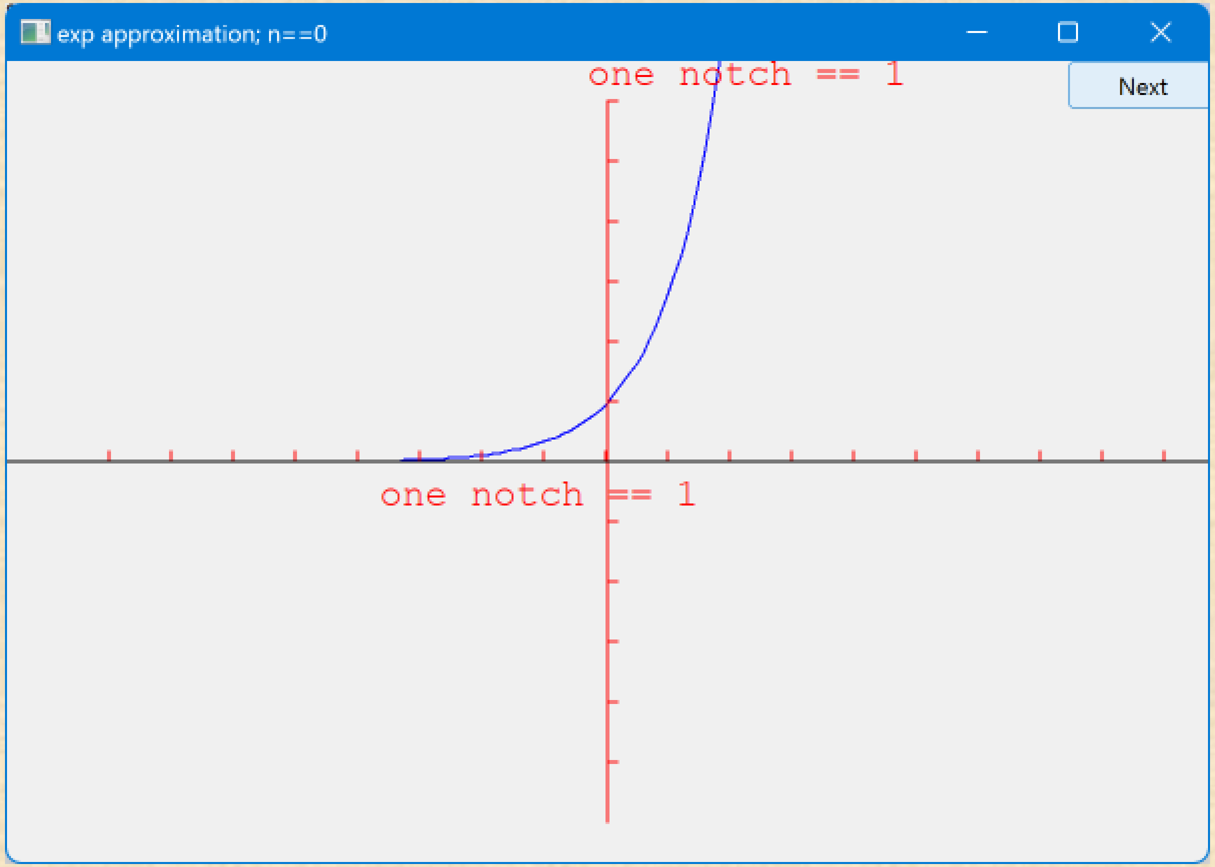
- Lambda expressions are important in conemporary C++
  - A shorthand notation for defining function objects

Demo

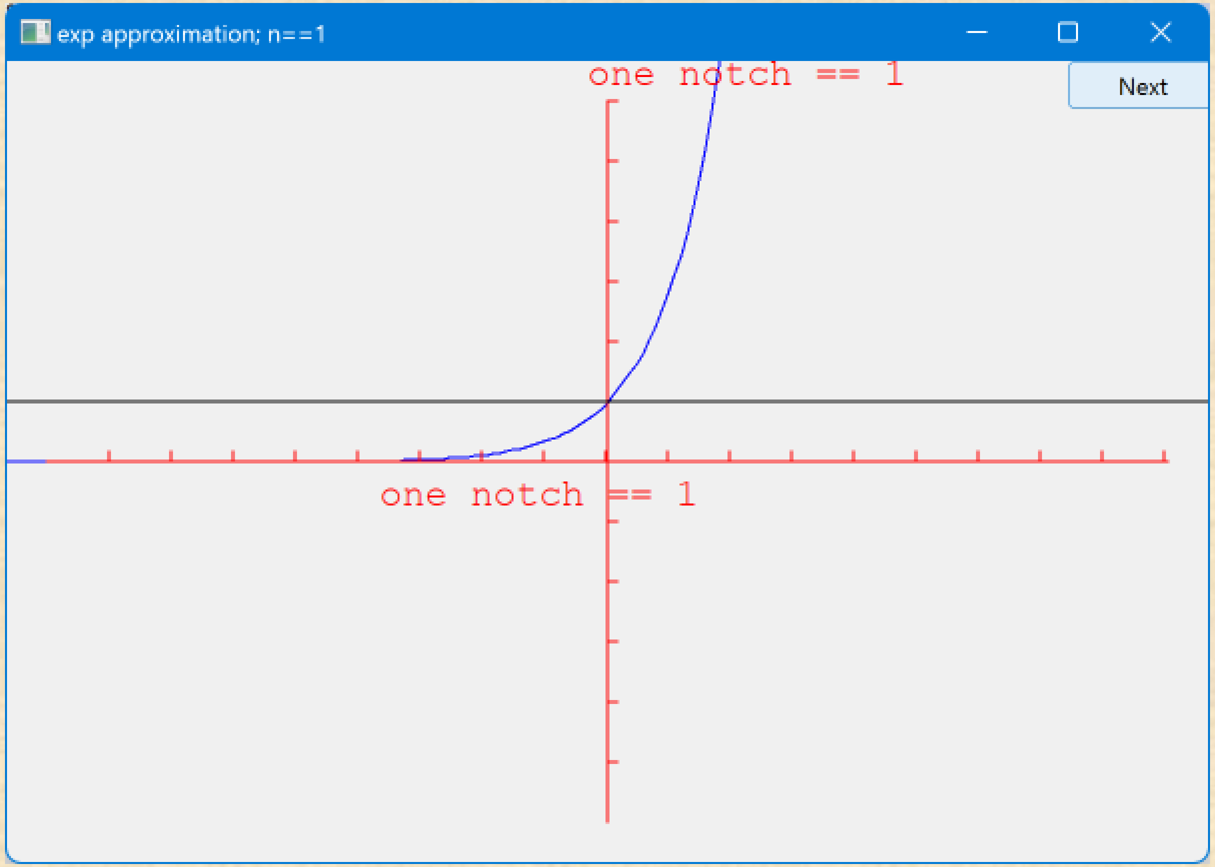


- The following screenshots are of the successive approximations of **exp(x)** using **exp_n(x,n)**

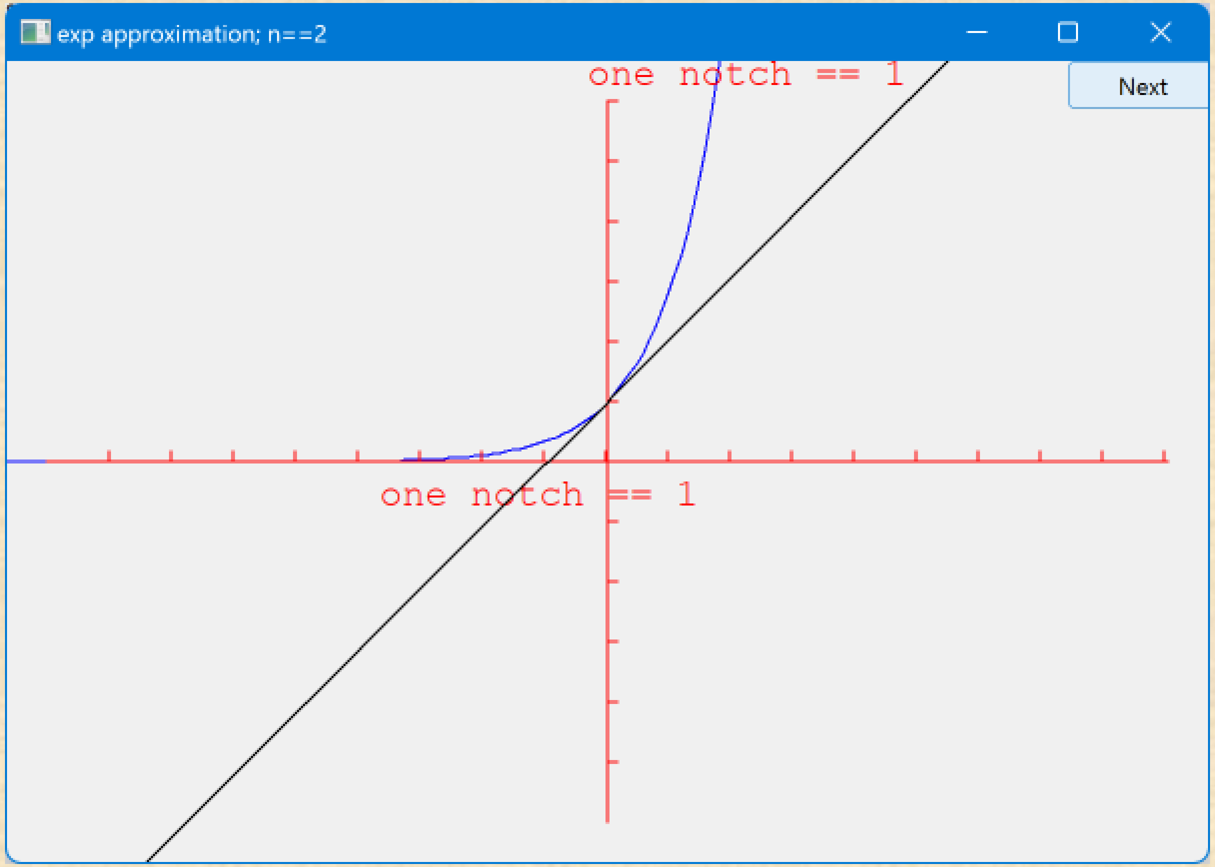
Demo: n = 0



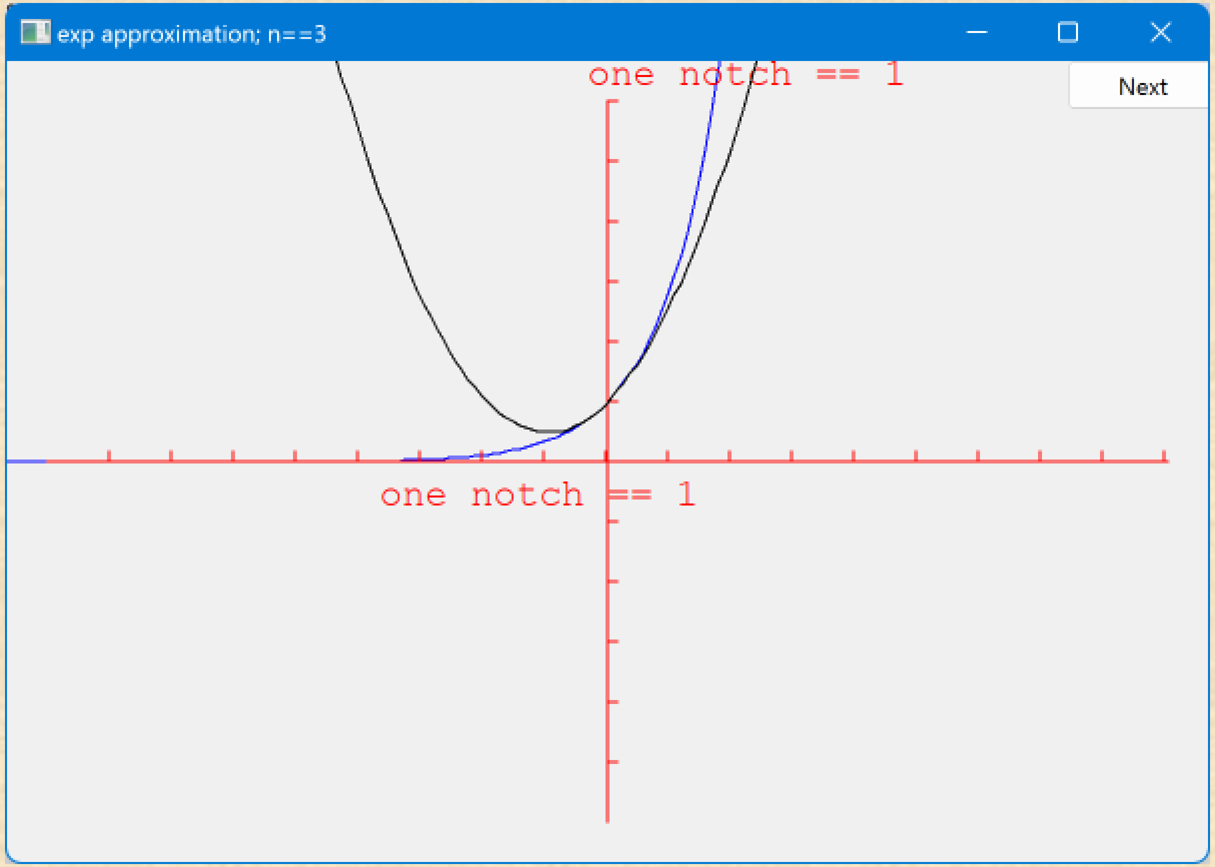
Demo: n = 1



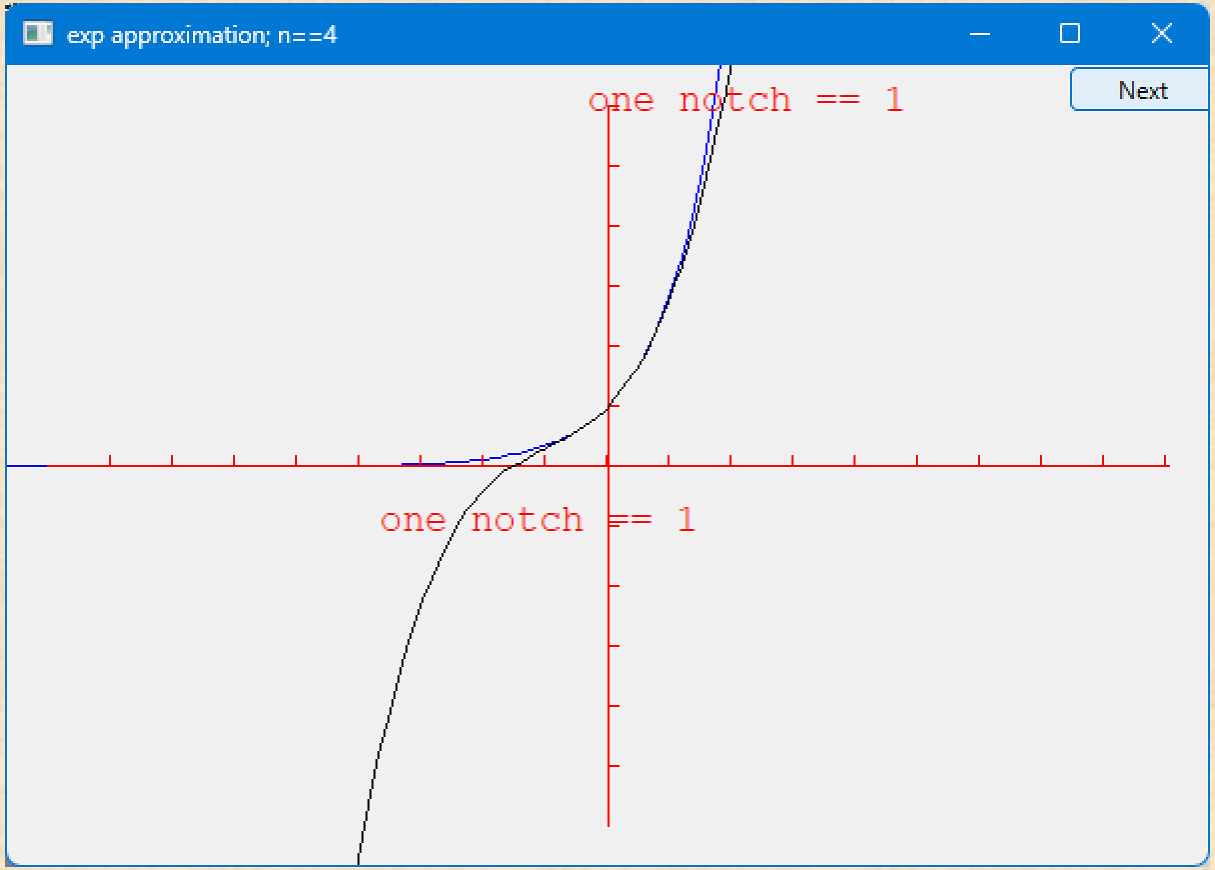
Demo: n = 2



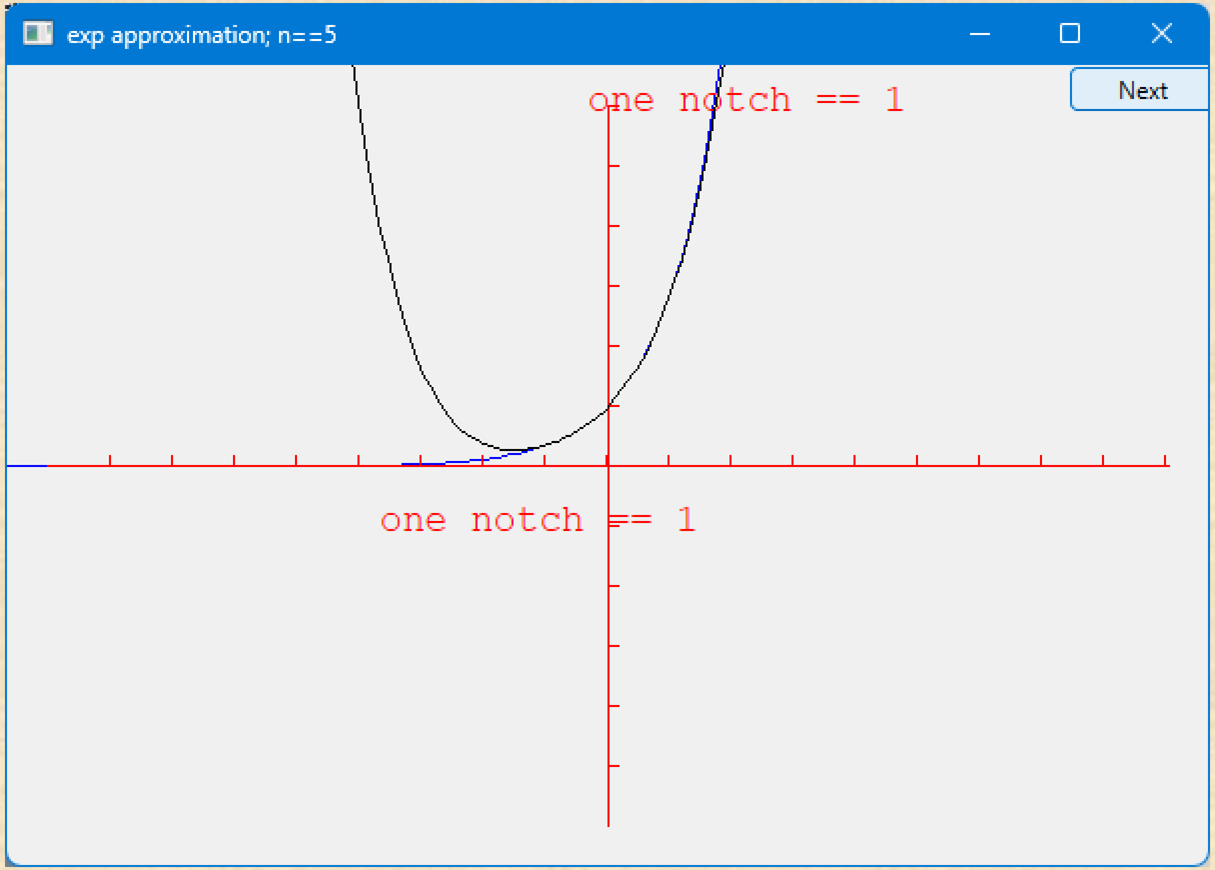
Demo: n = 3



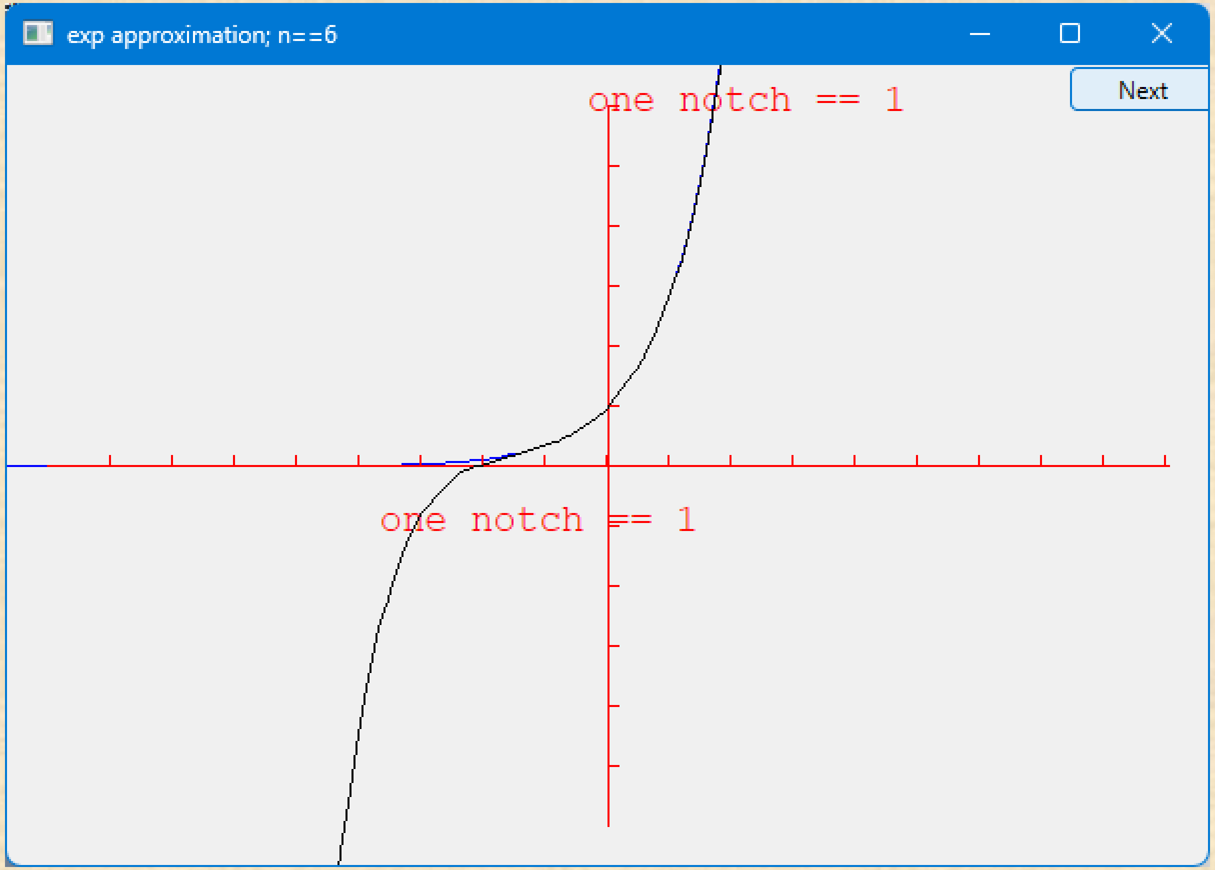
Demo: n = 4



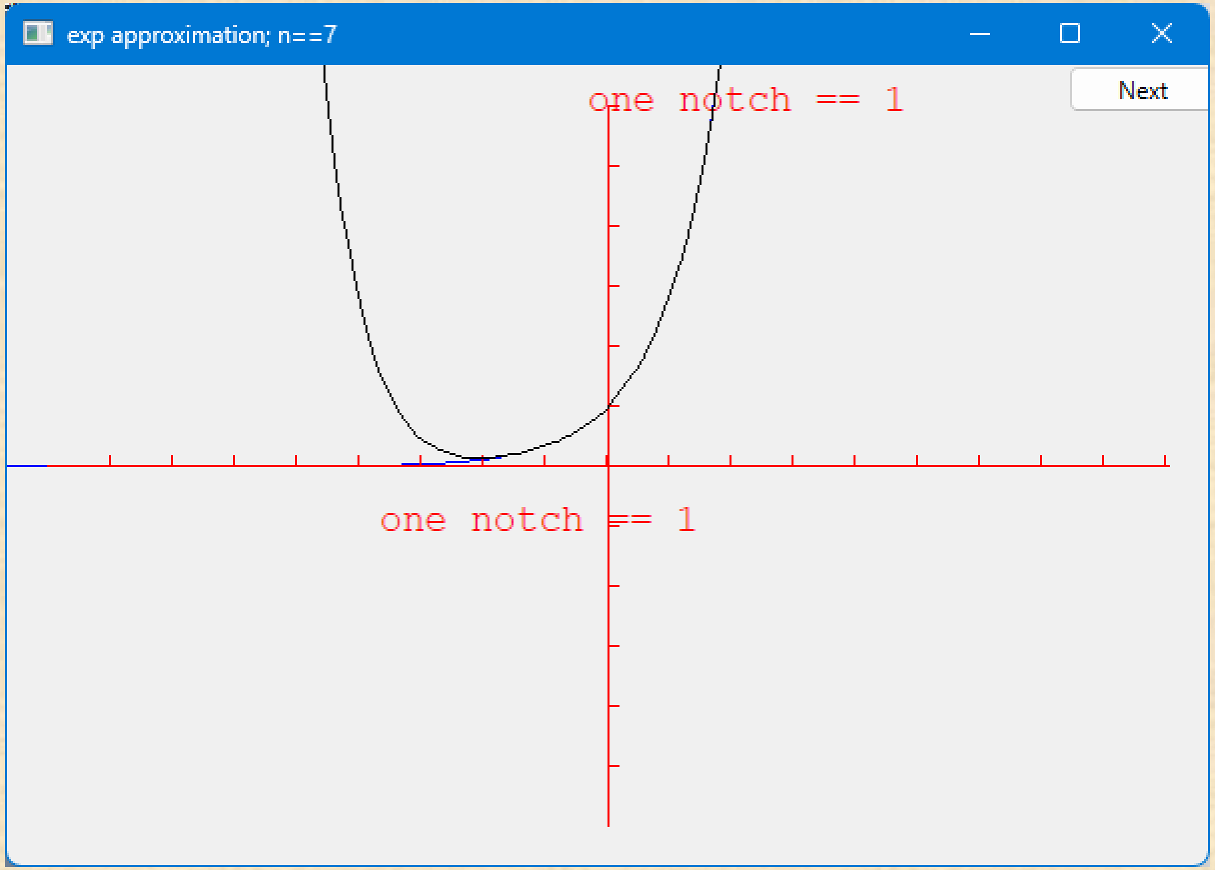
Demo: n = 5



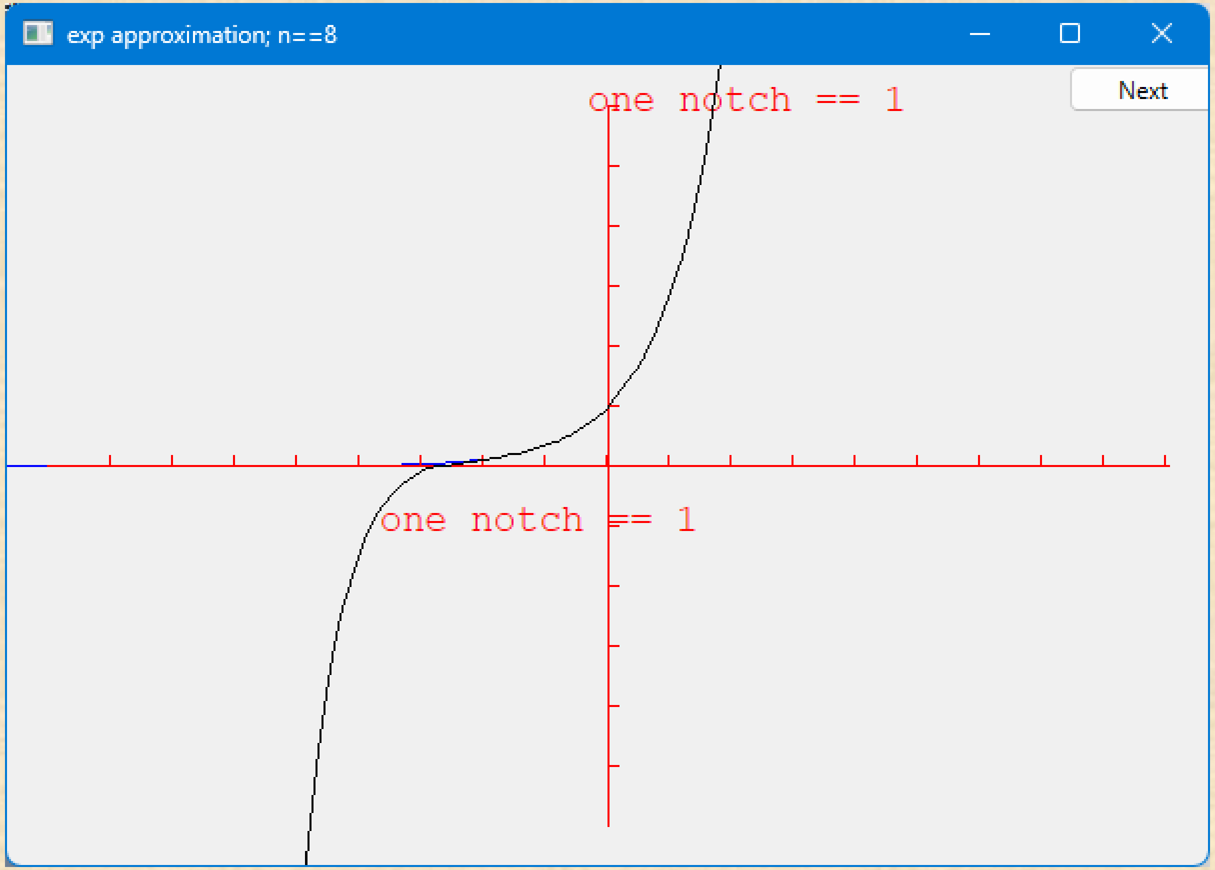
Demo: n = 6



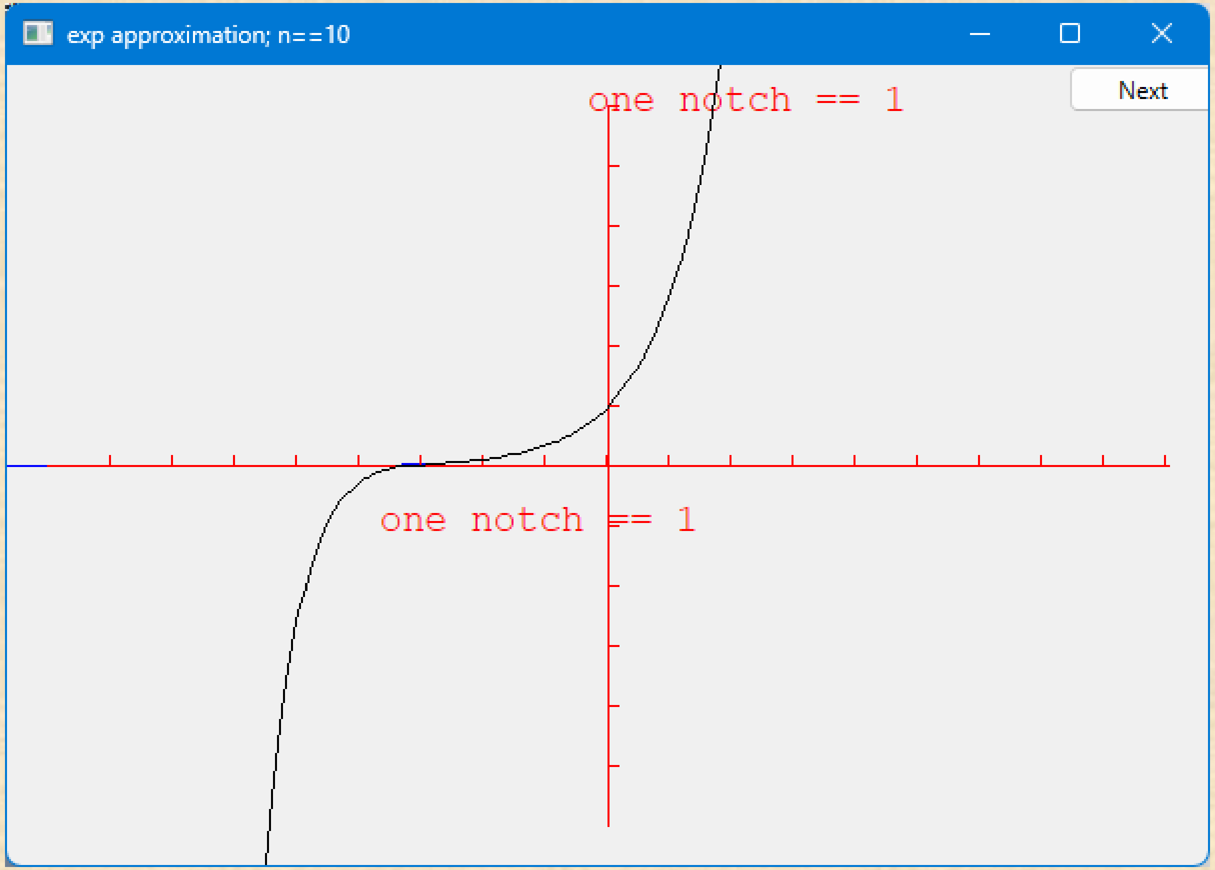
Demo: n = 7



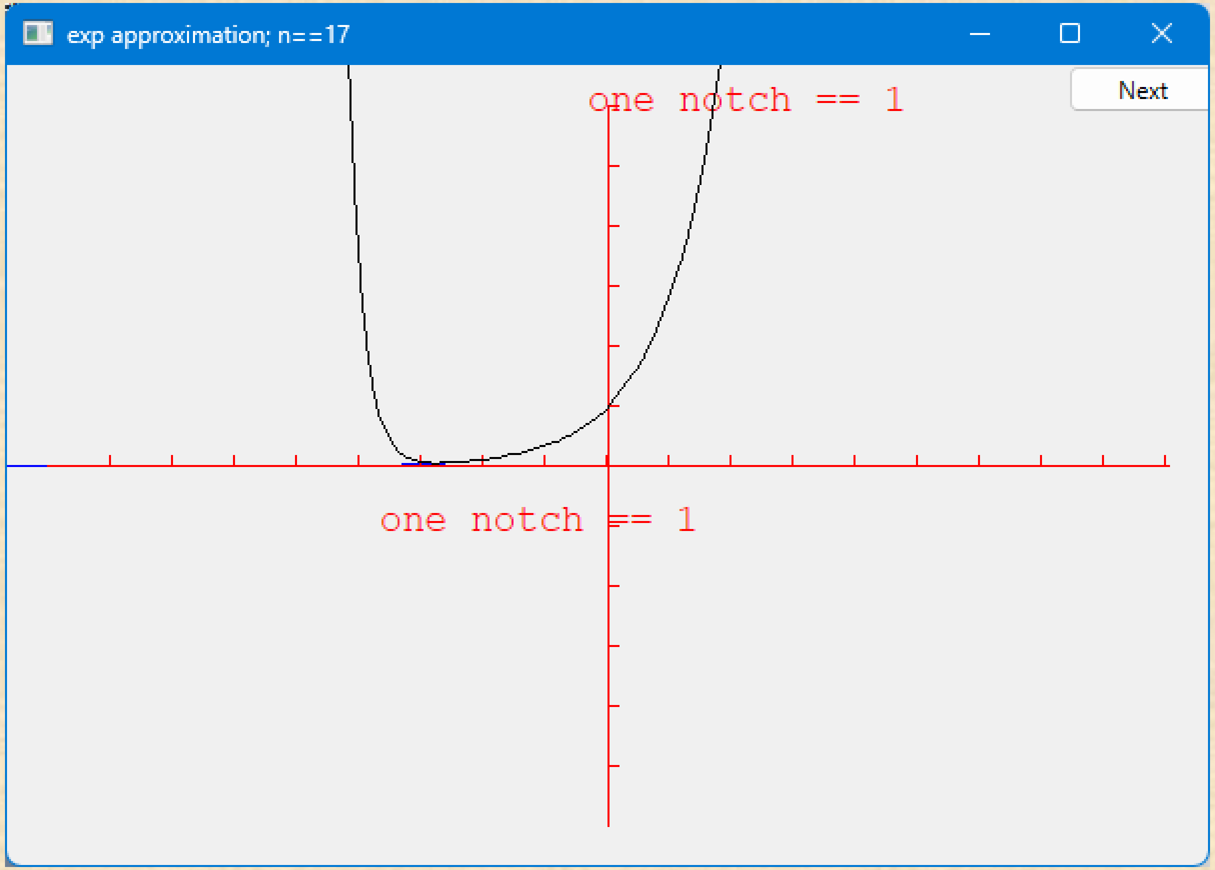
Demo: n = 8



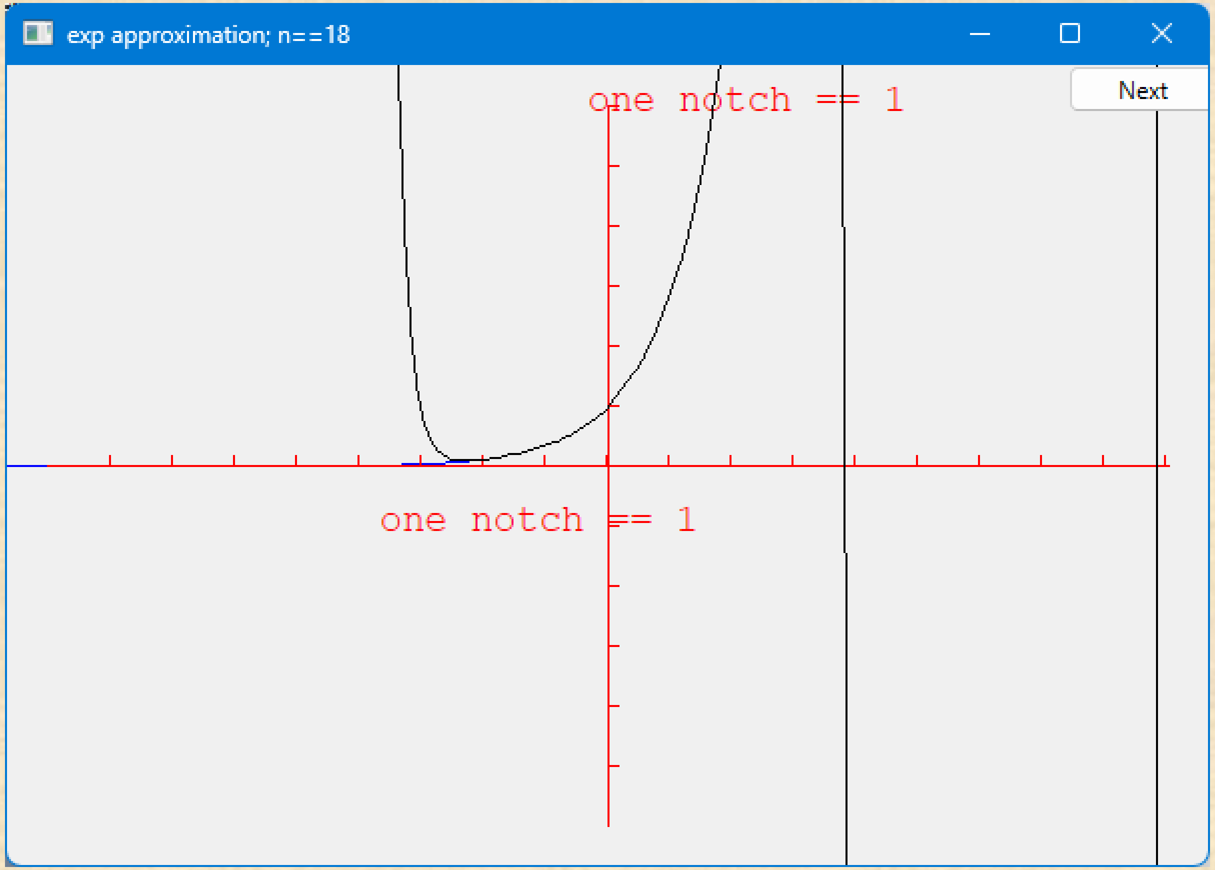
Demo: n = 10



Demo: n = 17

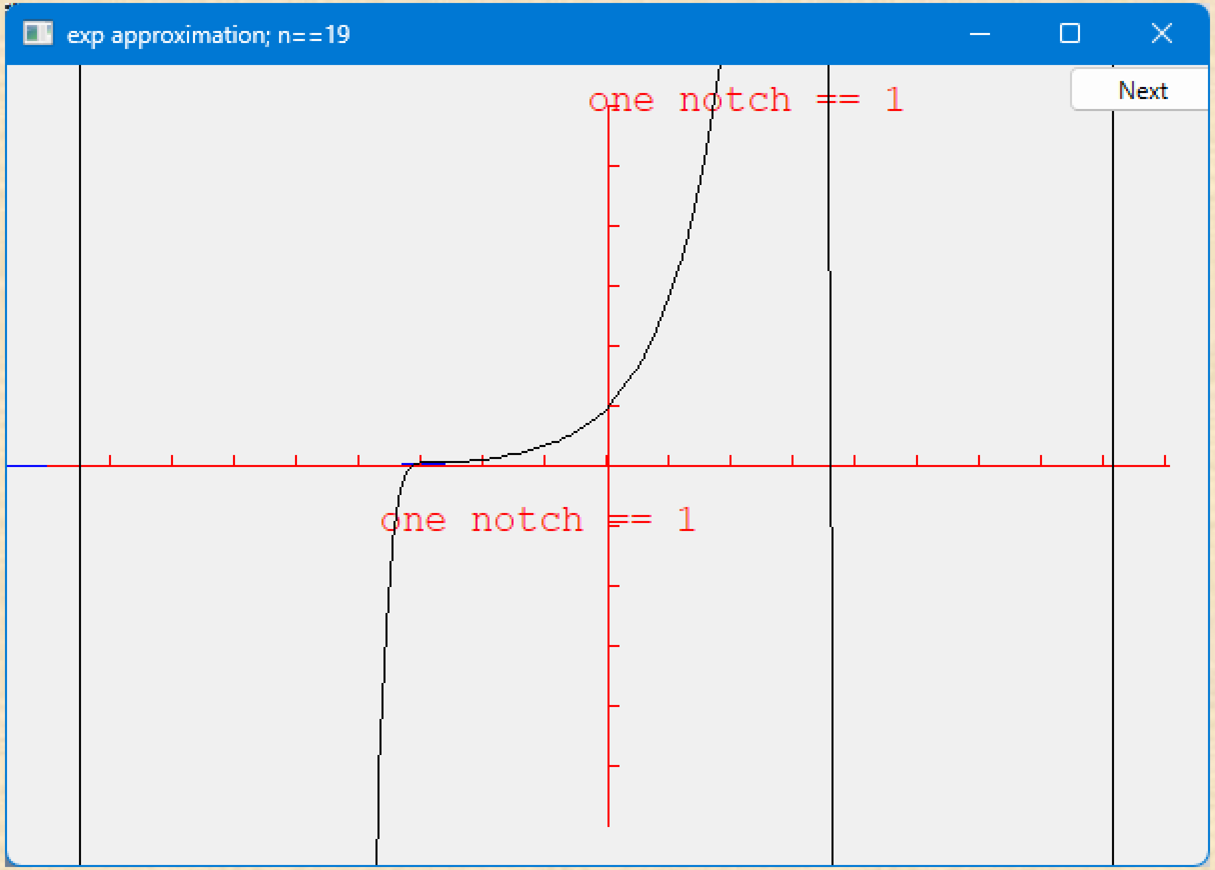


Demo: n = 18

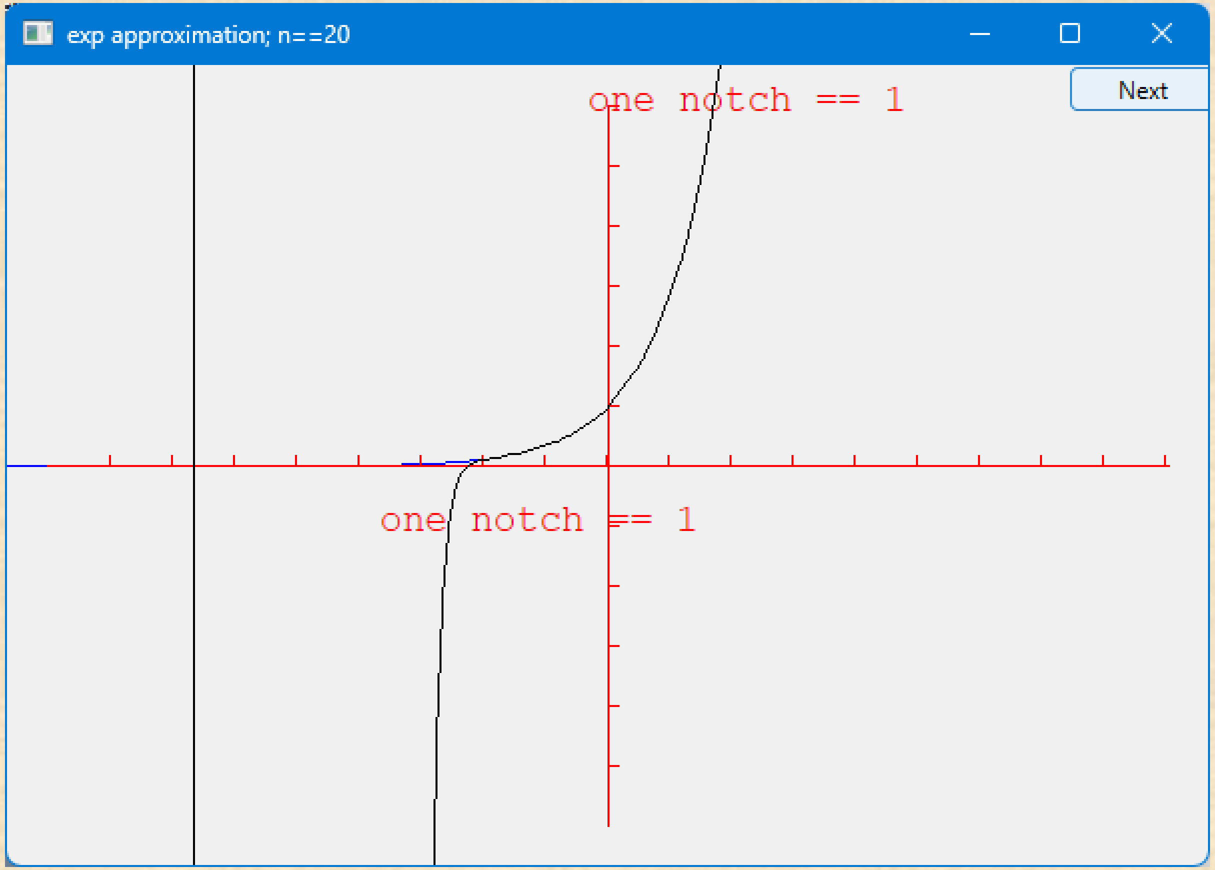


- Huh? Vertical lines???

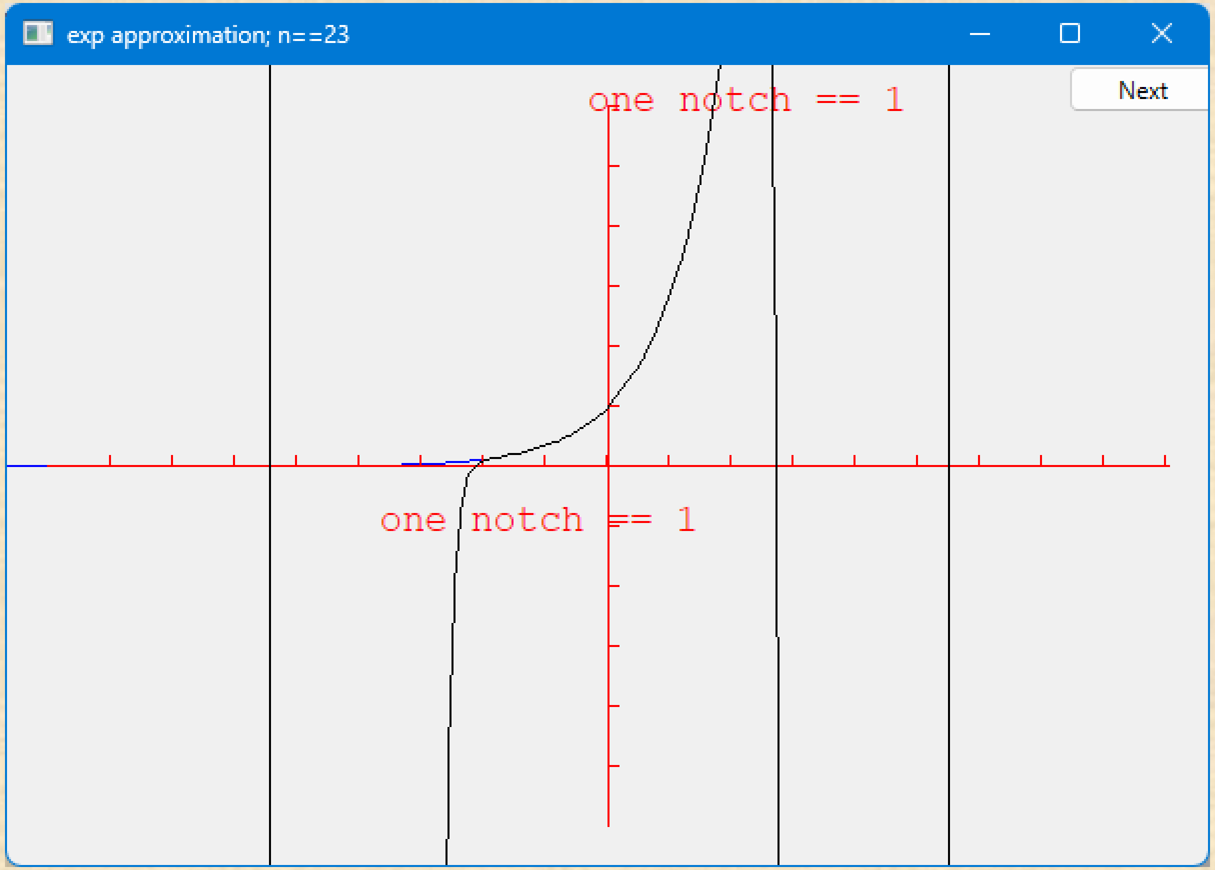
Demo: n = 19



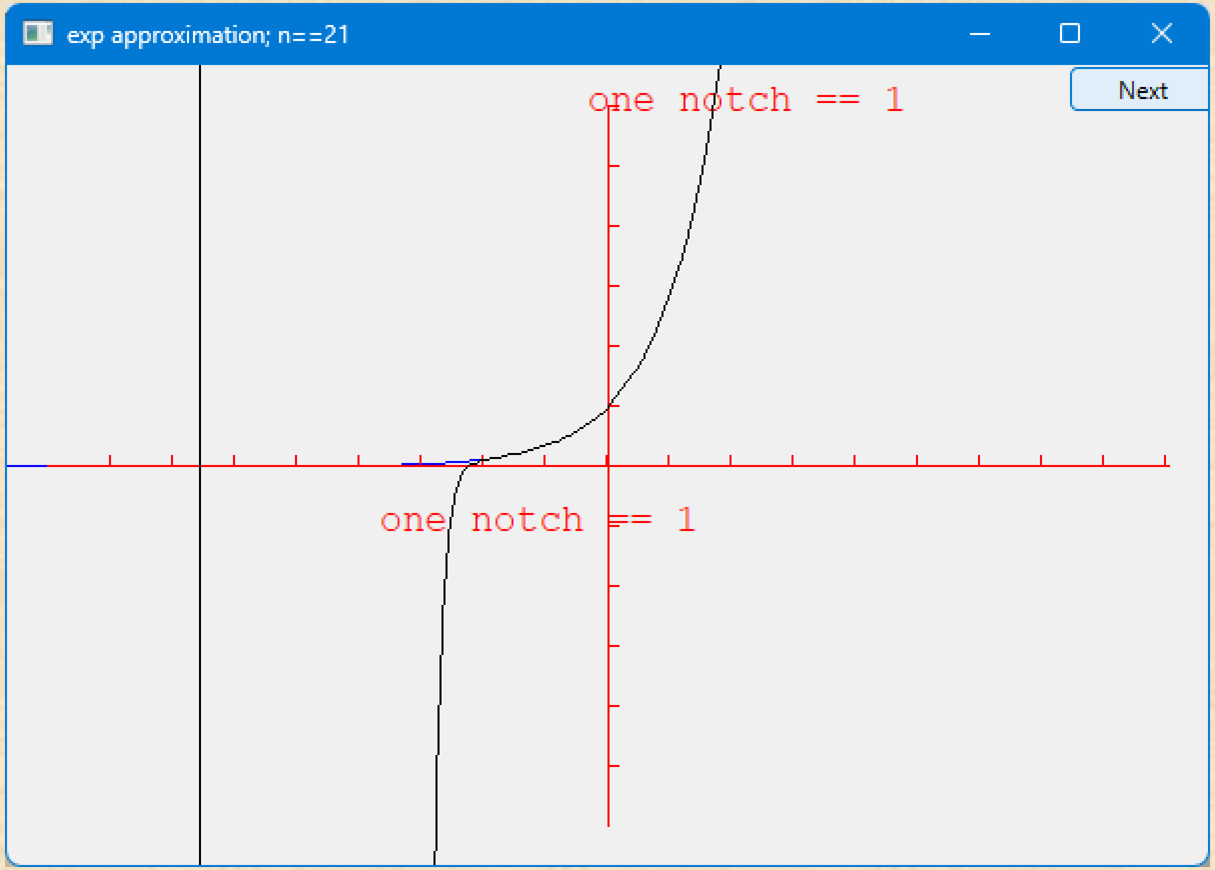
Demo: n = 20



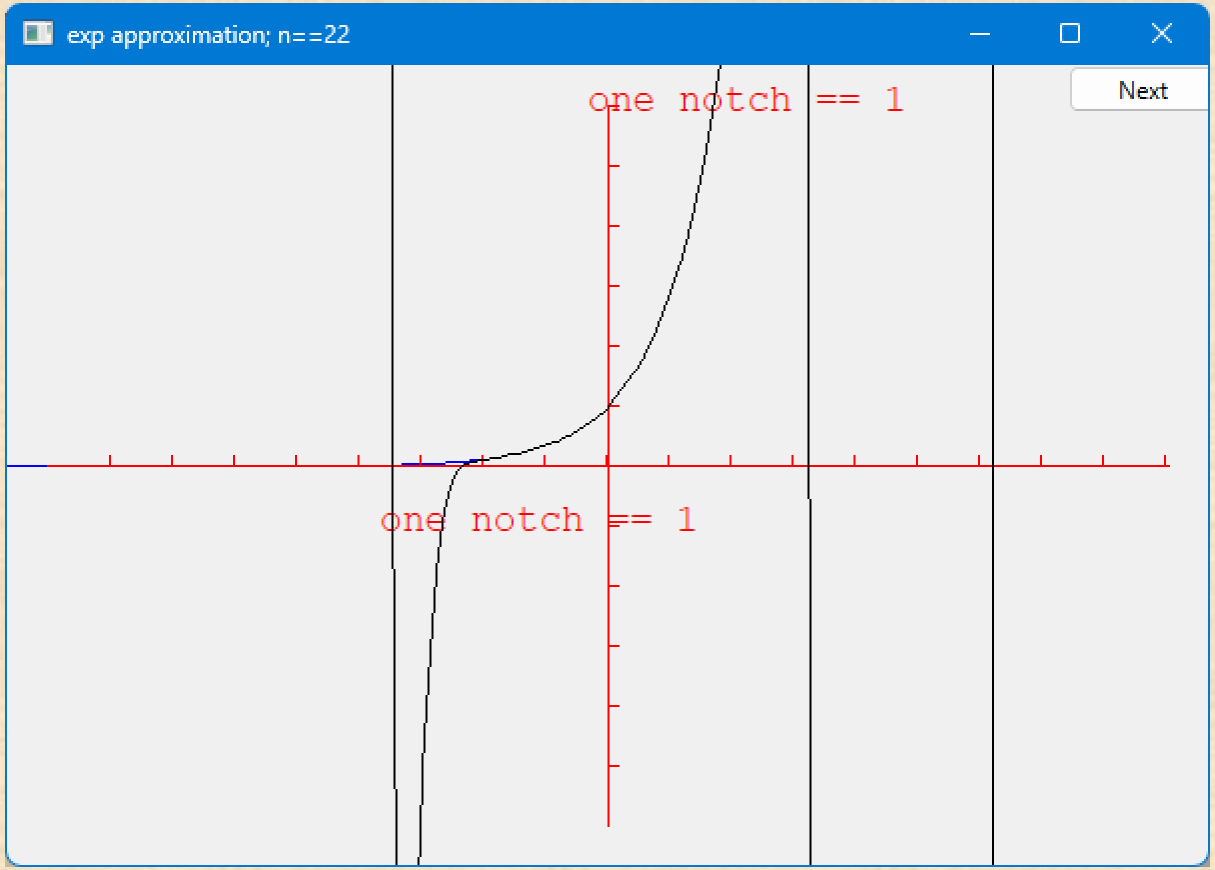
Demo: n = 21



Demo: n = 22



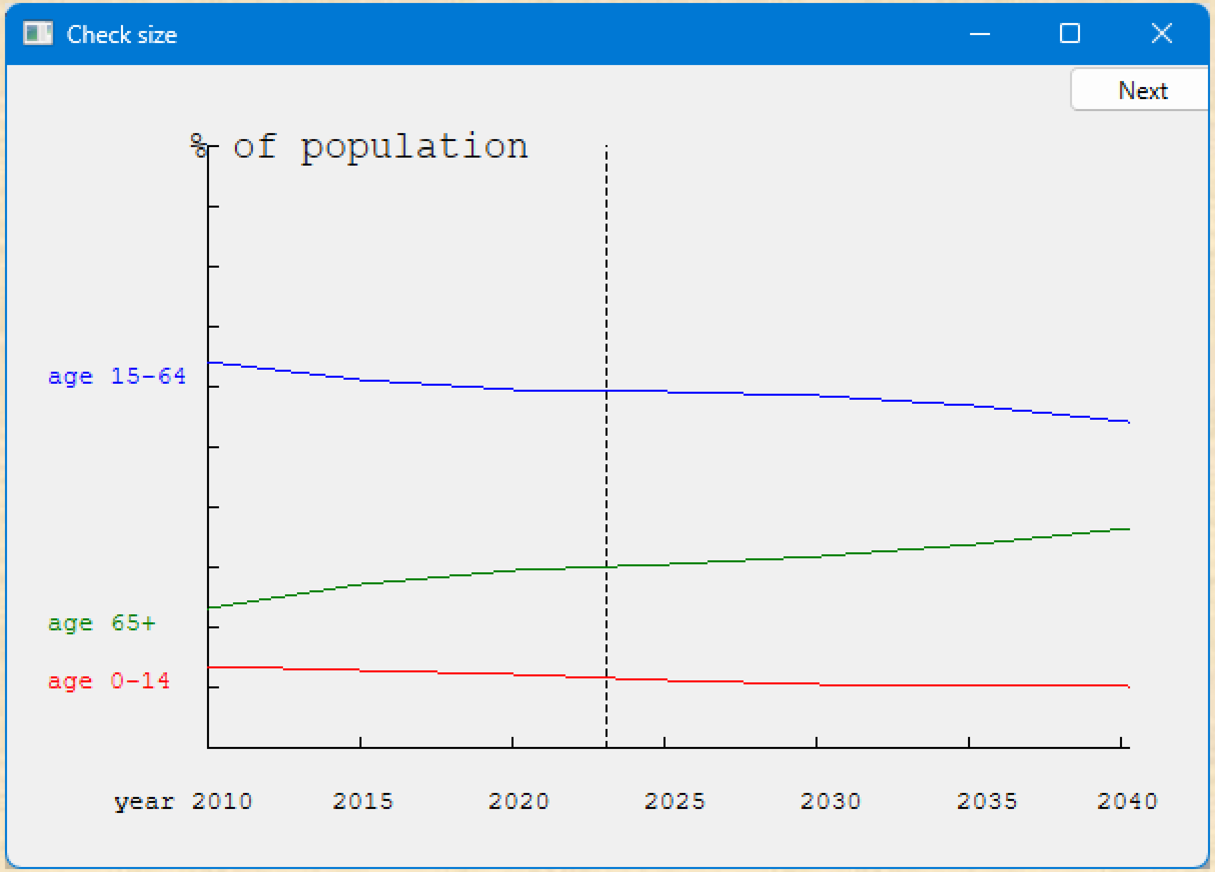
Demo: n = 23



Why did the graph “go wild”?

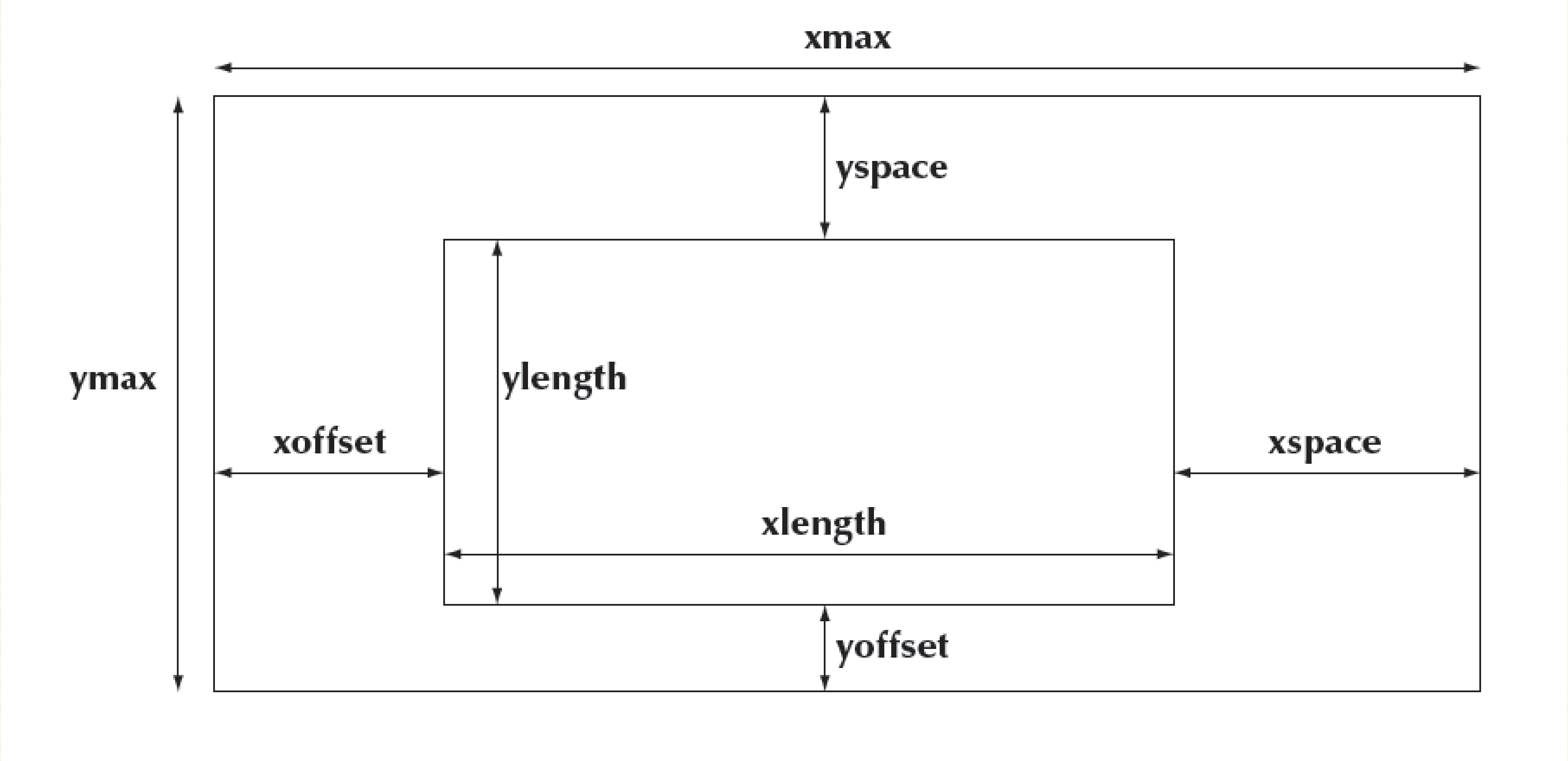
- Floating-point numbers are an approximations of real numbers
  - Just approximations
    - In a fixed amount of memory
  - Real numbers can be arbitrarily large and arbitrarily small
    - Floating-point numbers are of a fixed size and can’t hold all real numbers
  - Sometimes the approximation is not good enough for what you do
  - Small inaccuracies (rounding errors) can build up into huge errors
- Always
  - be suspicious about calculations
    - always
  - check your results
    - Visual representations of values can be most useful
  - hope that your errors are obvious
    - You want your code to break early – before anyone else gets to use it

Graphing data



- Often, what we want to graph is data, not a well-defined mathematical function
  - Here, we use three **Open_polylines**

Graphing data



- Carefully design your screen layout

Code for Axis

```
struct Axis : Shape {
        enum Orientation { x, y, z };
        Axis(Orientation d, Point xy, int length,
                int number_of_notches = 0,          // default: no notches
                string label = ""                   // default : no label
                );

        void draw_specifics(Painter& painter) const override;
        void move(int dx, int dy) override;
        void set_color(Color c);

        Text label;
        Lines notches;
        Line line;
};
```

```
Axis::Axis(Orientation d, Point xy, int length, int n, string lab)
        :label(Point{0,0},lab),
        line(xy,  (d==x ) ? Point(xy.x+length, xy.y)  :  Point(xy.x, xy.y-length))          // horizontal or
    vertical
{
        if (length<2) error("bad axis length");
        switch (d) {
        case Axis::x:
        {
                // ...
                break;
        }
        case Axis::y:
        {
                // ...
                break;
        }
        case Axis::z:
                error("z axis not implemented");
}
```

```
case Axis::x:
{
        int dist = length/n;
        int x = xy.x+dist;
        for (int i = 0; i<n; ++i) {
                notches.add(Point{x,xy.y},Point{x,xy.y-5});
                x += dist;
        }
        label.move(length/3,xy.y+20);        // label under the line
        break;
}
```

Axis implementation

```
void Axis::draw_specifics(Painter& painter) const
{
        line.draw(painter);        // the line
        notches.draw(painter);     // the notches may have a different color from the line
        label.draw(painter);       // the label may have a different color from the line
}
```

- The underlying Qt implementation slightly shines through

Axis implementation

```
void Axis::move(int dx, int dy)
{
        Shape::move(dx,dy);        // the line
        notches.move(dx,dy);
        label.move(dx,dy);
        redraw();
}

void Axis::set_color(Color c)
{
        Shape::move(dx,dy);
        notches.move(dx,dy);
        label.move(dx,dy);
        redraw();
}
```

Next Lecture

- Graphical user interfaces
- Windows and Widgets
- Buttons and dialog boxes