USTC

Chapter 15
Functions and graphing

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 本章，我们提供一种绘制函数图和数据图的方法，以及一些相关的编程技术

Note

- 本课程是关于程序设计的
  - 我们提供的示例－如图形化－是下面的简单例子
    - 有用的编程技术
    - 构建实用程序的有用工具

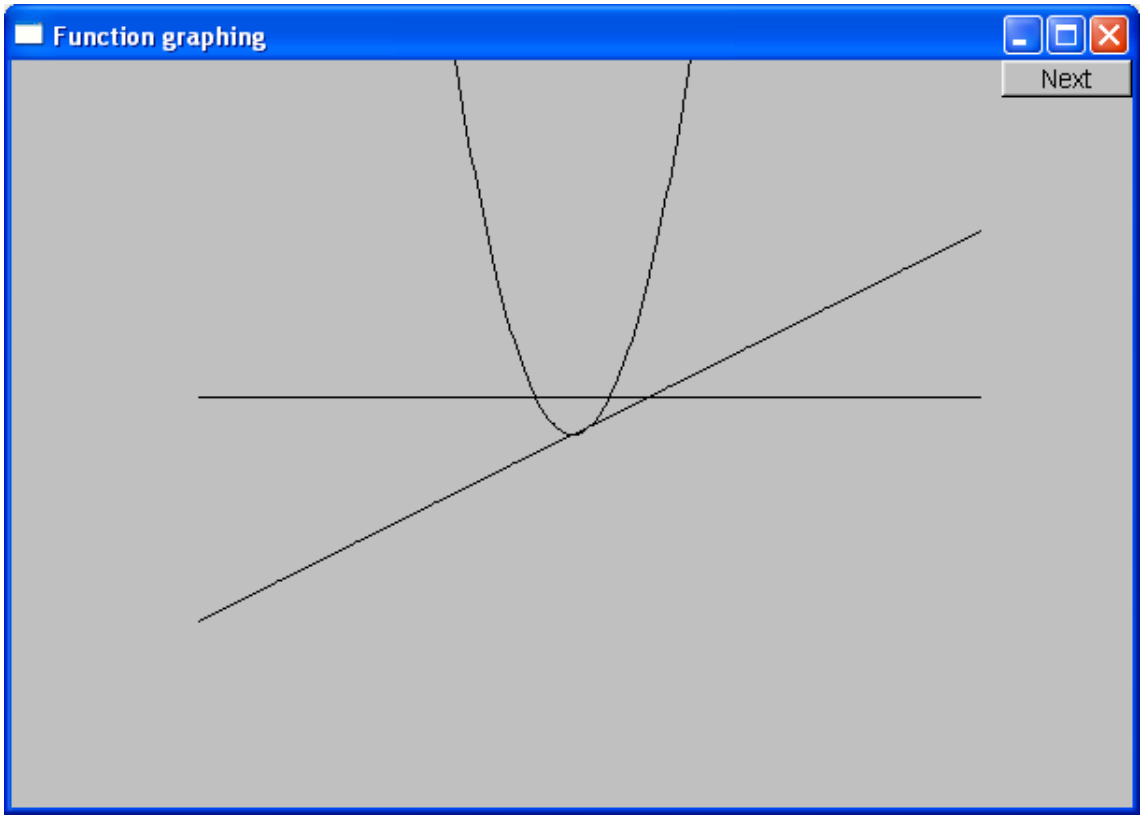
    通过我们的示例考虑它们的构造方法
  - “大问题”是如何被分解为小问题并进行分别处理的？
  - 类是如何被定义和使用的？
    - 它们是否有合理的数据成员？
    - 它们是否有有用的成员函数？
  - 变量的使用
    - 太少？
    - 太多？
    - 你如何更好的命名它们？

绘制函数图

- 从简单的例子开始
  - 总是记住 “Hello, World!” 的作用
- 绘制的函数是输入单个参数输出单个值的
  - 绘制点 (x, f(x))，x 值在给定的范围内 [r1, r2)
- 让我们首先绘制三个简单的函数

```
double one(double x) { return 1; }          // y==1
double slope(double x) { return x/2; }      // y==x/2
double square(double x) { return x*x; }     // y==x*x
```

函数




```
double one(double x) { return 1; }          // y==1
double slope(double x) { return x/2; }      // y==x/2
double square(double x) { return x*x; }     // y==x*x
```

How do we write code to do this?

需要绘制的函数

```
Simple_window win0(Point(100,100),xmax,ymax,"Function graphing");

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);

win0.attach(s);
win0.attach(s2);
win0.attach(s3);

win0.wait_for_button( );
```

图形绘制的 x 范围

为了使图形合理地显示在窗口中（单位问题）

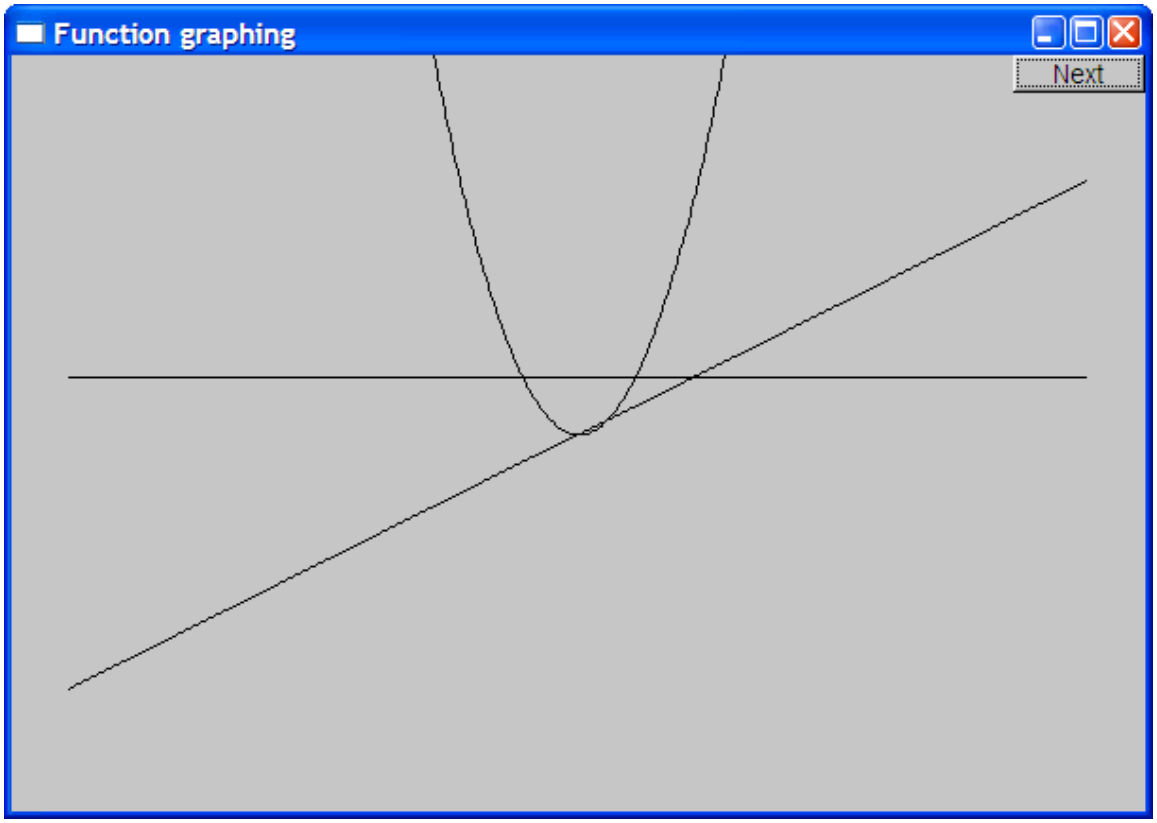
我们需要一些常量

```
const int xmax = 600;                    // window size
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 (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 = 20;                  // scaling  factors
const int y_scale = 20;
```

// Choosing a center (0,0), scales, and number of points can be fiddly

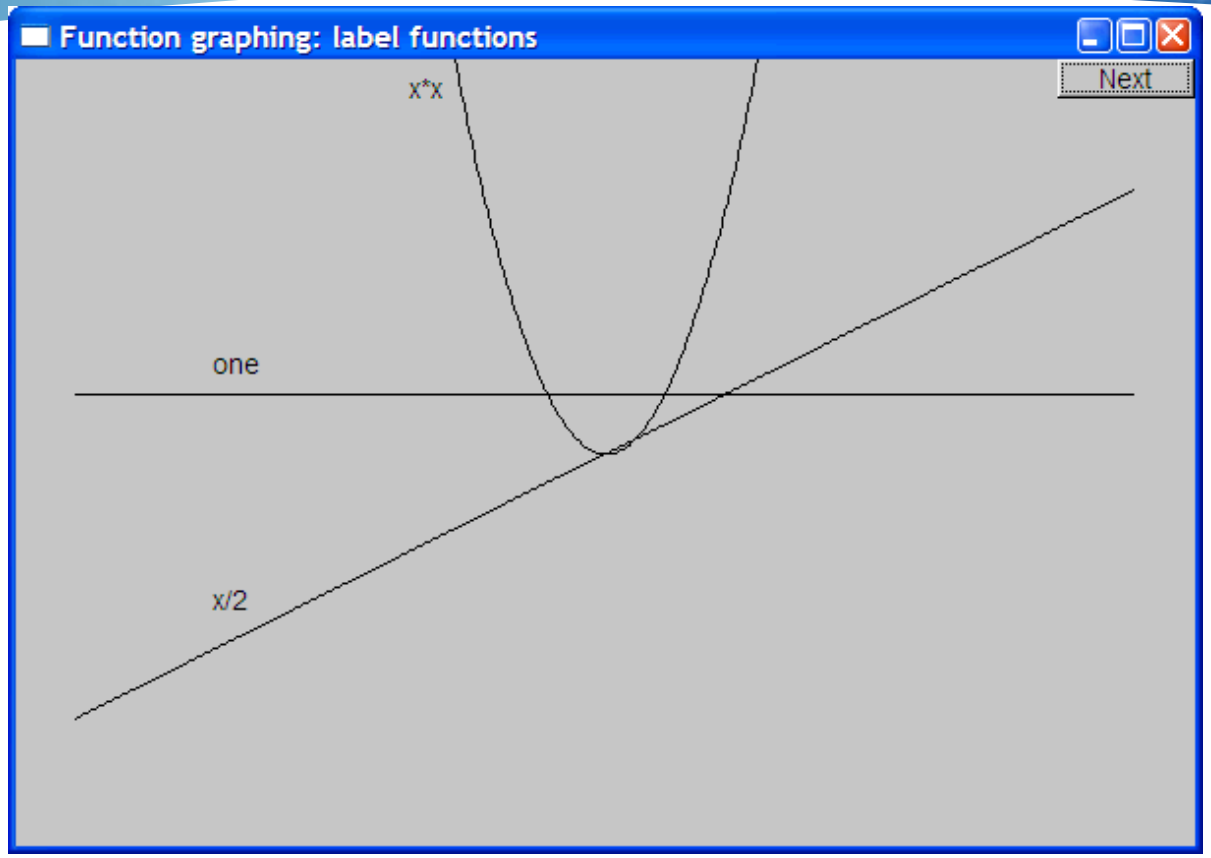
// The range usually comes from the definition of what you are doing

Function – 但它代表什么含义呢?



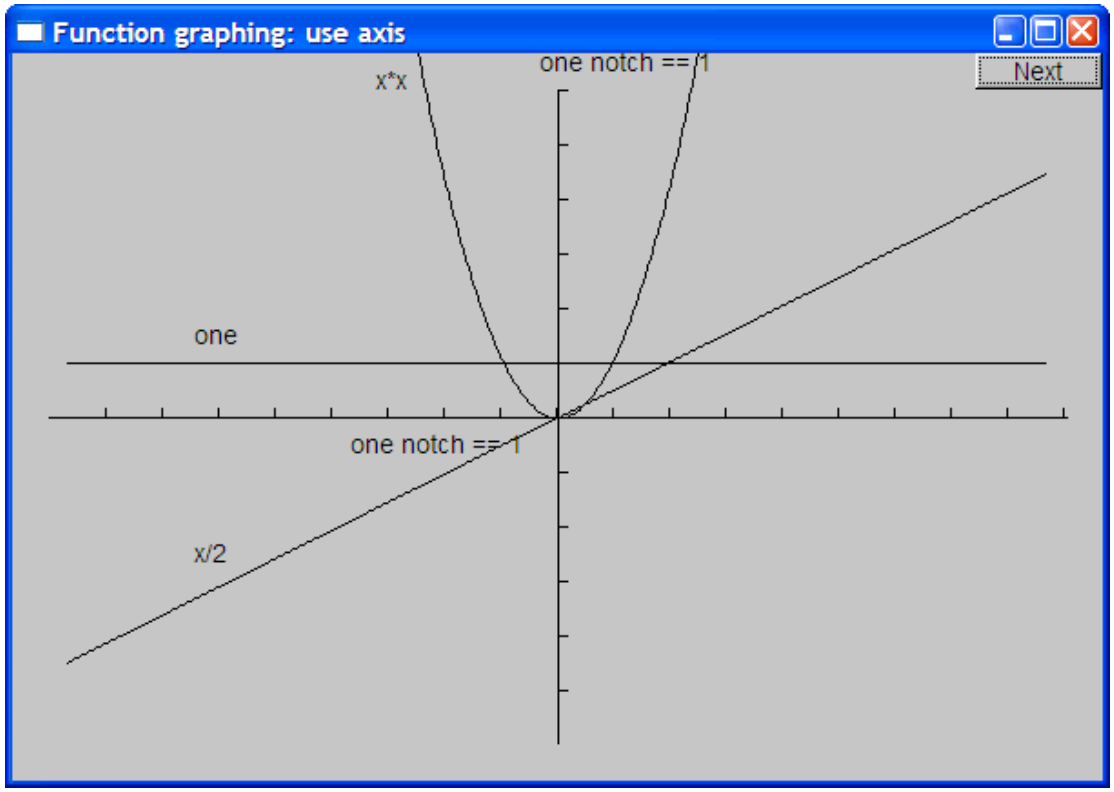
- 上图有什么问题?
  - 没有坐标轴(无刻度)
  - 没有标签

给函数曲线加上标签




```
Text ts(Point(100,y_orig-30),"one");
Text ts2(Point(100,y_orig+y_orig/2-10),"x/2");
Text ts3(Point(x_orig-90,20),"x*x");
```

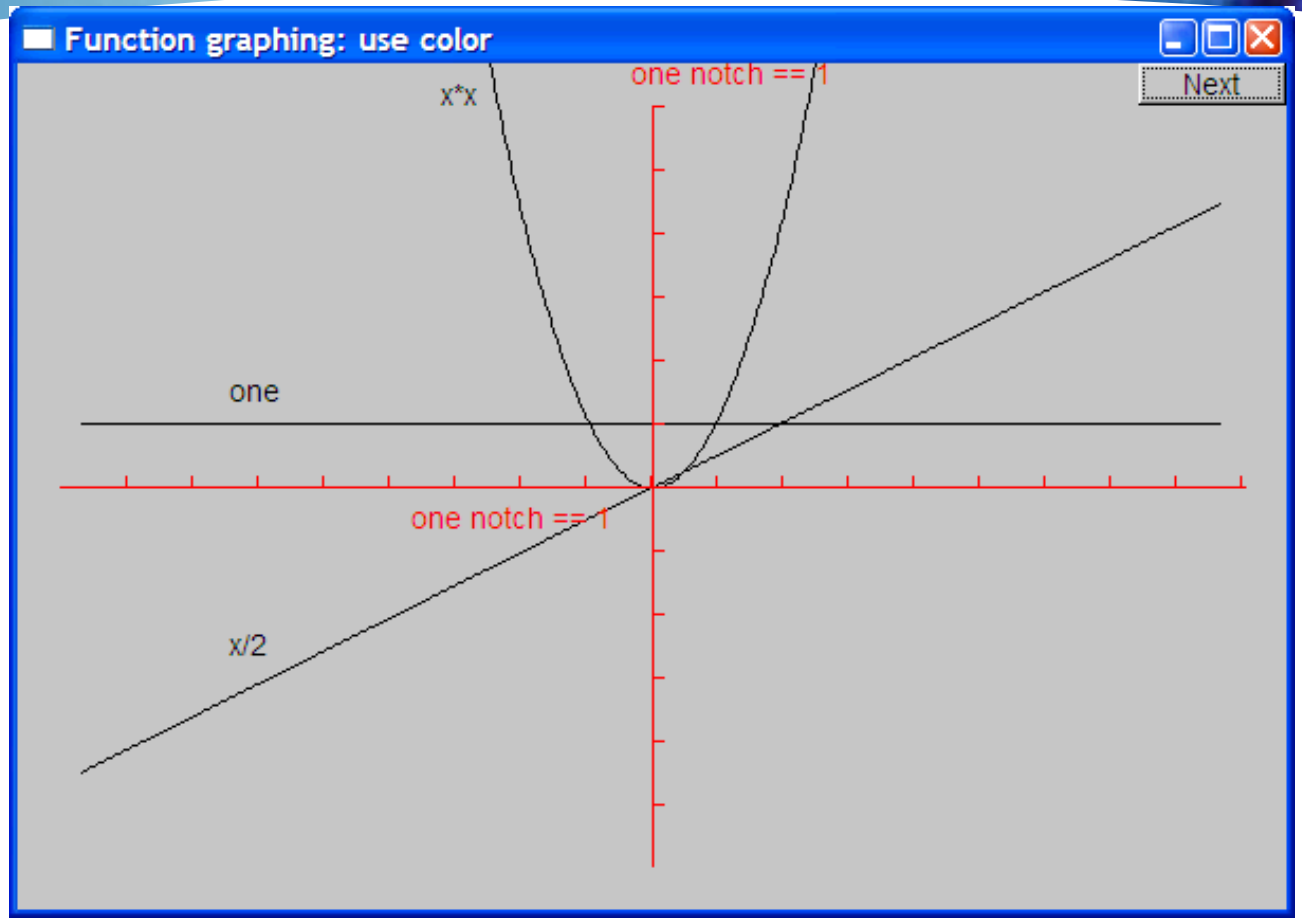
添加x和y坐标轴



- 使用坐标轴来显示原点(0,0)和刻度

```
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");
```

适量地使用颜色




```
s.set_color(Color::green);
y.set_color(Color::red);
```

```
x.set_color(Color::red);
ts.set_color(Color::green);
```

Function 的实现

- 需要一个类型来指定函数参数(绘制哪个函数)
  - typedef 用于为一个类型定义一个新名称(别名)
    - **typedef int Color;** *// now Color means int*
  - 定义我们需要的参数类型，Fct
    - **typedef double Fct(double);** *// now Fct means function*
      // taking a double argument
      // and returning a double

传统的函数指针定义

  - 函数类型 Fct 举例：

```
double one(double x) { return 1; }      // y==1
double slope(double x) { return x/2; } // y==x/2
double square(double x) { return x*x; }          // y==x*x
```

现在定义“Function”

```
struct Function : Shape                    // Function is derived from Shape
{
    // all it needs is a constructor:
    Function(
        Fct f,                  // f is a Fct  (takes a double, returns a double)

        double r1,  // the range of x values (arguments to f) [r1:r2)
        double r2,
        Point orig, // the screen location of (0,0)
        int count,              // number of points used to draw the function
                                // (number of line segments used is count-1)

        double xscale ,         // the location (x,f(x)) is (xscale*x,yscale*f(x))
        double yscale
    );
};
```

Function 的实现

```
Function::Function( Fct f,
                    double r1, double r2,        // range
                    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;
    }
}
```

前向检查

Function并不存储传递给它的构造函数参数值

默认参数

- 前面示例中，七个参数太多了！
  - 在程序设计中，太多参数的情况是常见的
  - 我们可能会迷惑或发生错误
  - 对一些参数提供默认值
    - 只能将末尾的参数定义为默认参数
    - 默认参数对构造函数往往比较有用

```
struct Function : Shape {
    Function( Fct f, double r1, double r2, Point xy,
            int 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
```

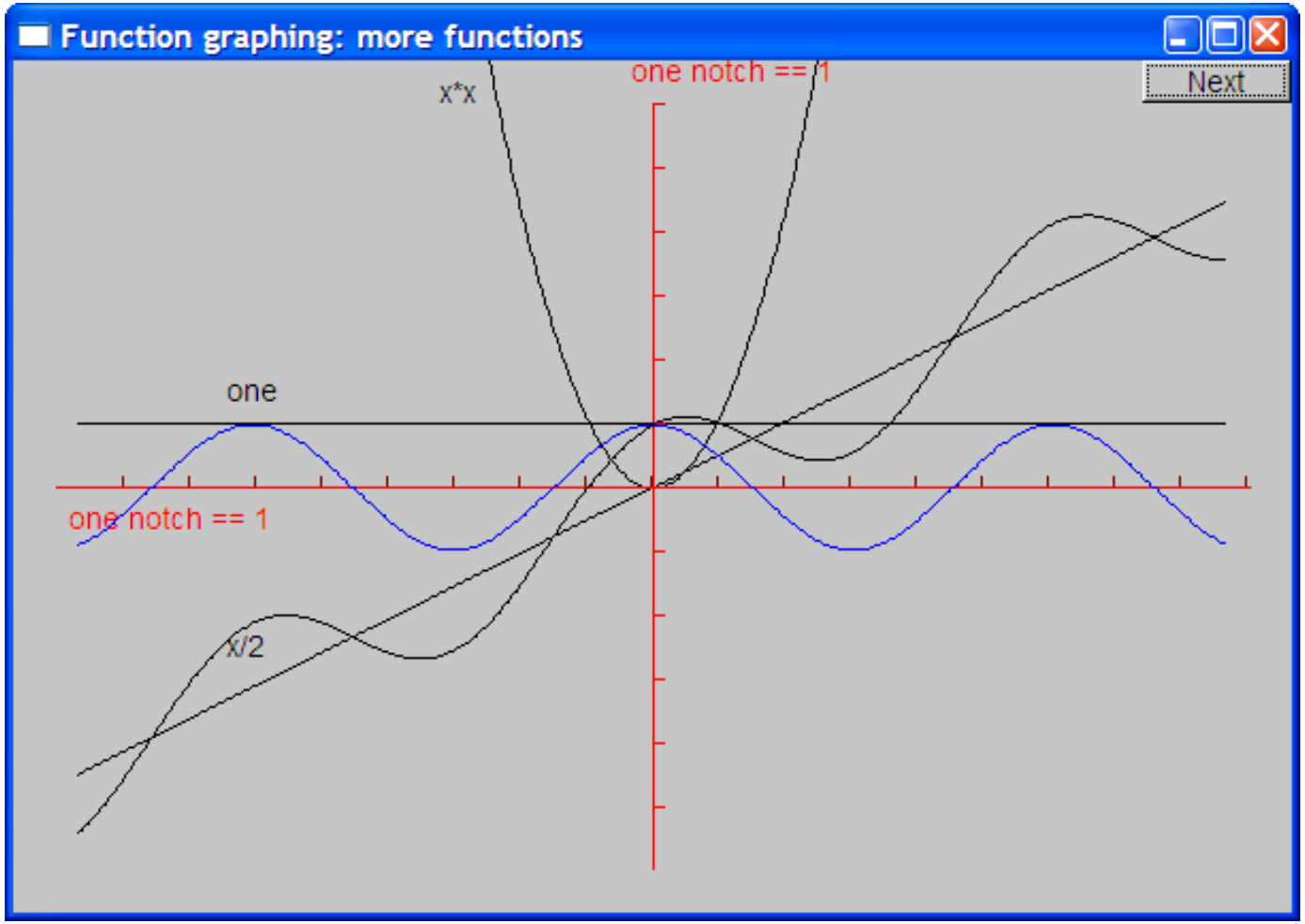
Function

- **Function 是一个好的类吗？**
  - No
    - Why not? —— 函数形式约束、近似等
  - 如何使用好这些点和缩放参数？
    - 参见 15.6.3
  - 如果你不能很好的完成它，就简化它，这样用户能够做他们需要的事情
    - 比如：增加参数是调用者能够更好地控制
    - 默认参数提供了一种有效的折中处理方式

更多函数

```
#include<cmath>          // standard mathematical functions

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

Function s4(cos,-10,11,orig,400,20,20);
s4.set_color(Color::blue);
Function s5(sloping_cos,-10,11,orig,400,20,20);
```

Cos 和 sloping-cos



标准数学函数 (<cmath>)

- **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);**

都是单double输入和单double输出

标准数据函数(<cmath>)

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

双输入和单输出

为什么需要绘制图形？

- 你能够在图形中看到一些从数字中无法看到的内容(直观)
  - 如果你从来没有看过 sine 曲线，你如何理解它？
- 可视化
  - 在很多领域中是理解的关键
  - 在大多数科研和商业领域都会用到
    - 科学、医学、商务、电信、大系统控制等

另一个示例：e^x

$$e^x == 1 + x + x^2/2! + x^3/3! + x^4/4! + x^5/5! + x^6/6! + x^7/7! + \ldots$$

此处，！表示阶乘 (e.g. 4!==4*3*2*1)

近似 e^x 的简单算法

```
double fac(int n) { /* ... */ }  // factorial ——阶乘

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

double expe(double x, int n) // sum of n terms of x
{
    double sum = 0;
    for (int i = 0; i<n; ++i) sum+=term(x,i);
    return sum;
}
```

近似 e^x 的简单算法

- 不过，我们目前只有单个参数的图形化函数，那么如何图形expe(x,n) 呢?

```
int expN_number_of_terms = 6;        // nasty sneaky argument to expN

double expN(double x)        // sum of expN_number_of_terms terms of x
{
    return expe(x,expN_number_of_terms);
}
```

一种不太好的解决方案：采用变量传值而不是参数传值

对 e^x 的近似

```
Simple_window win(Point(100,100),xmax,ymax,"");
// the real exponential :
Function real_exp(exp,r_min,r_max,orig,200,x_scale,y_scale);
real_exp.set_color(Color::blue);
win.attach(real_exp);

const int xlength = xmax-40;
const 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");

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

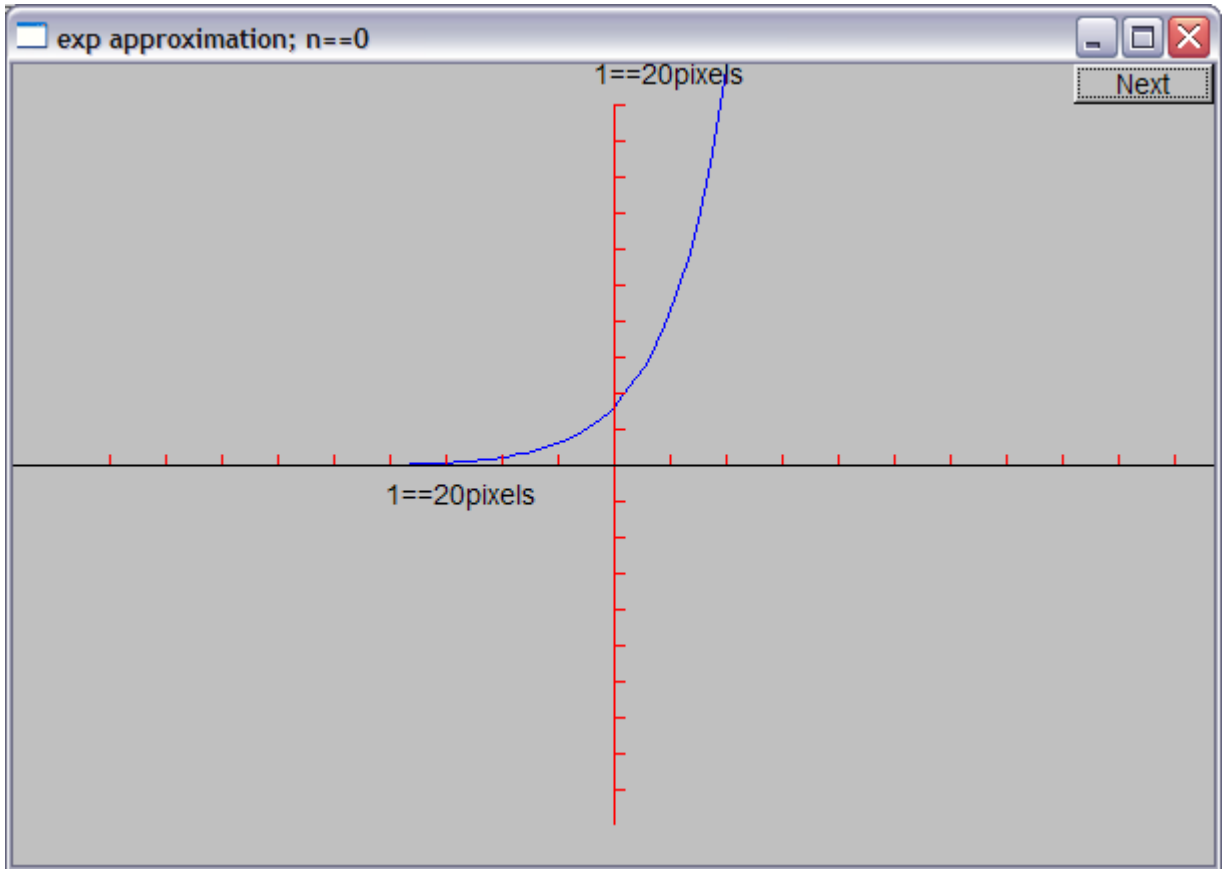
对 e^x 的近似

```
for (int n = 0; n<50; ++n) {
    ostringstream ss;
    ss << "exp approximation; n==" << n ;
    win.set_label(ss.str().c_str());
    expN_number_of_terms = n;        // nasty sneaky argument to expN

    // next approximation:
    Function e(expN,r_min,r_max,orig,200,x_scale,y_scale);

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

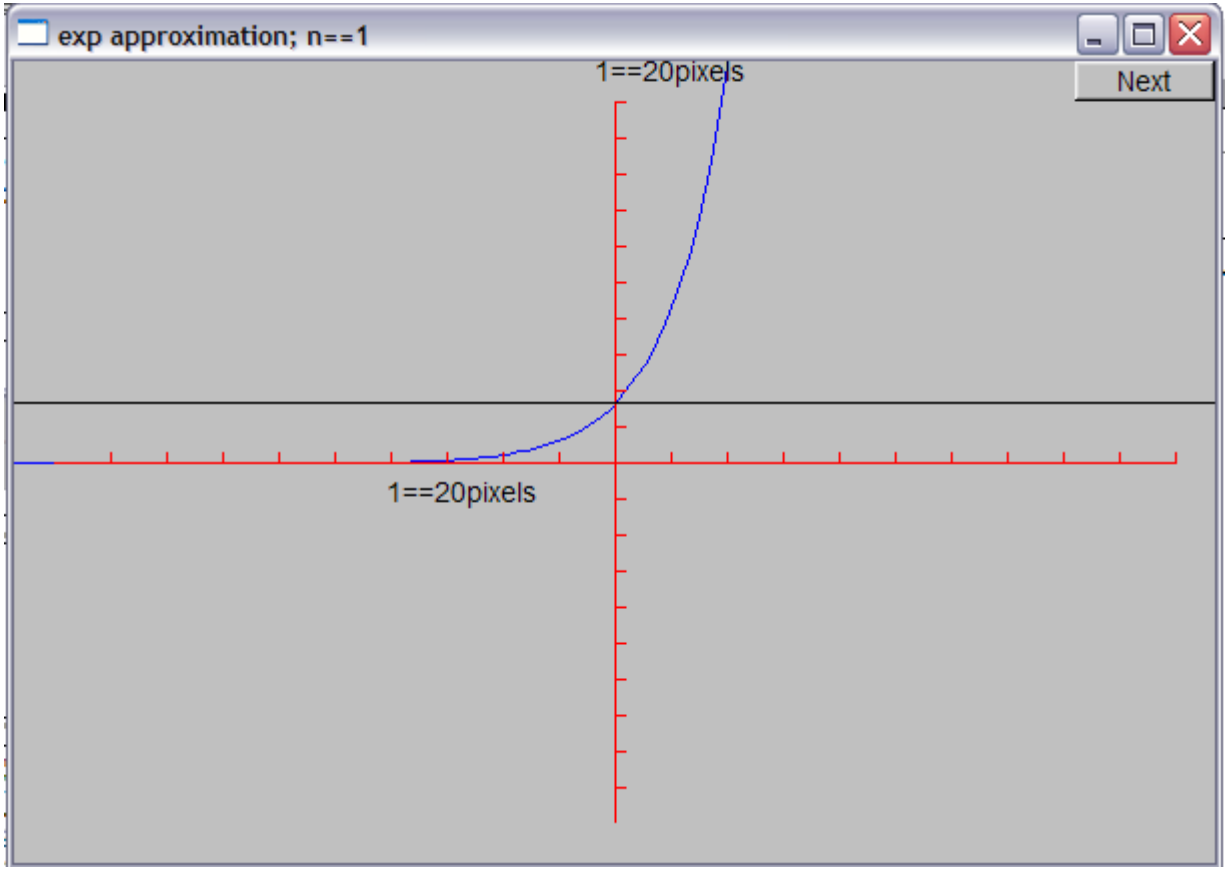
Demo

- 下面的截图是采用 expe(x,n) 对 exp(x) 的连续近似
  - 采用多少项能够很好的近似，即 n 的取值？

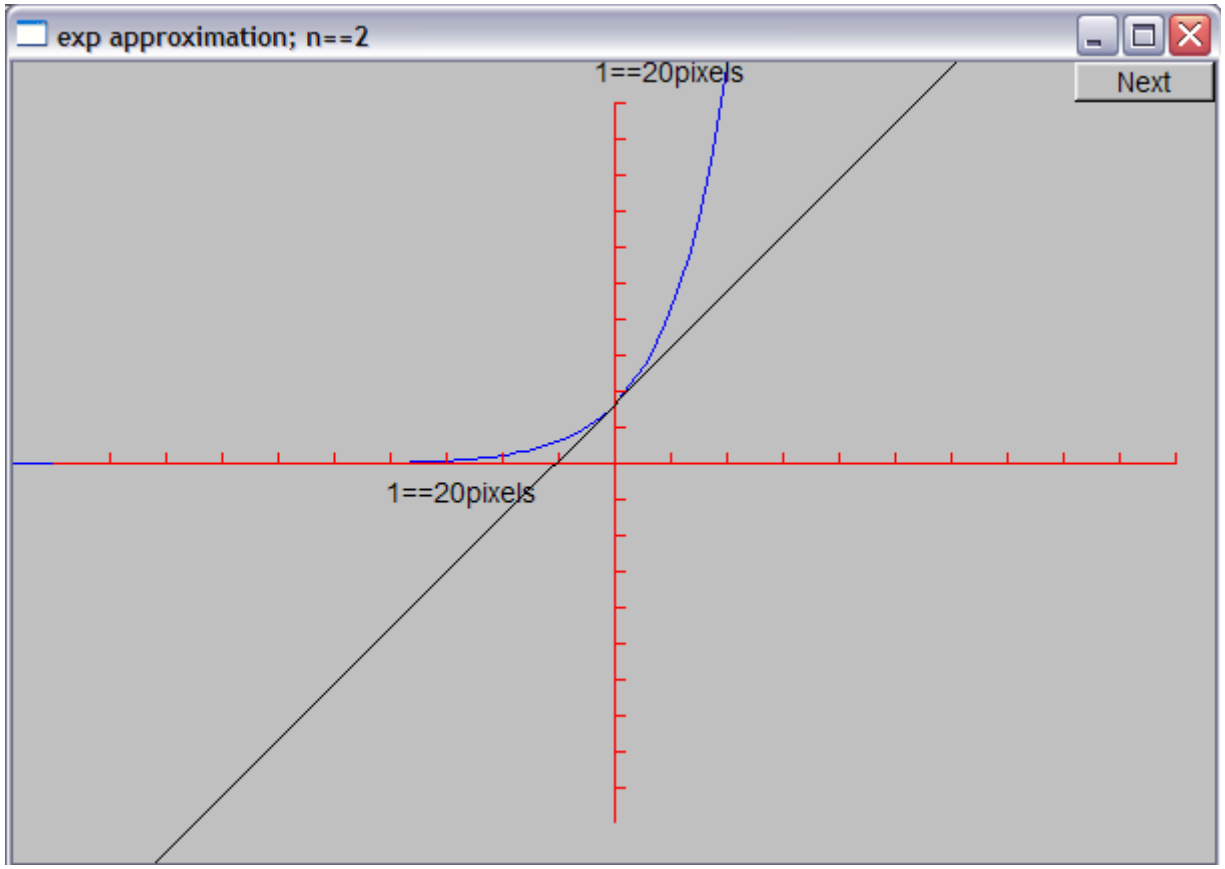
Demo n = 0



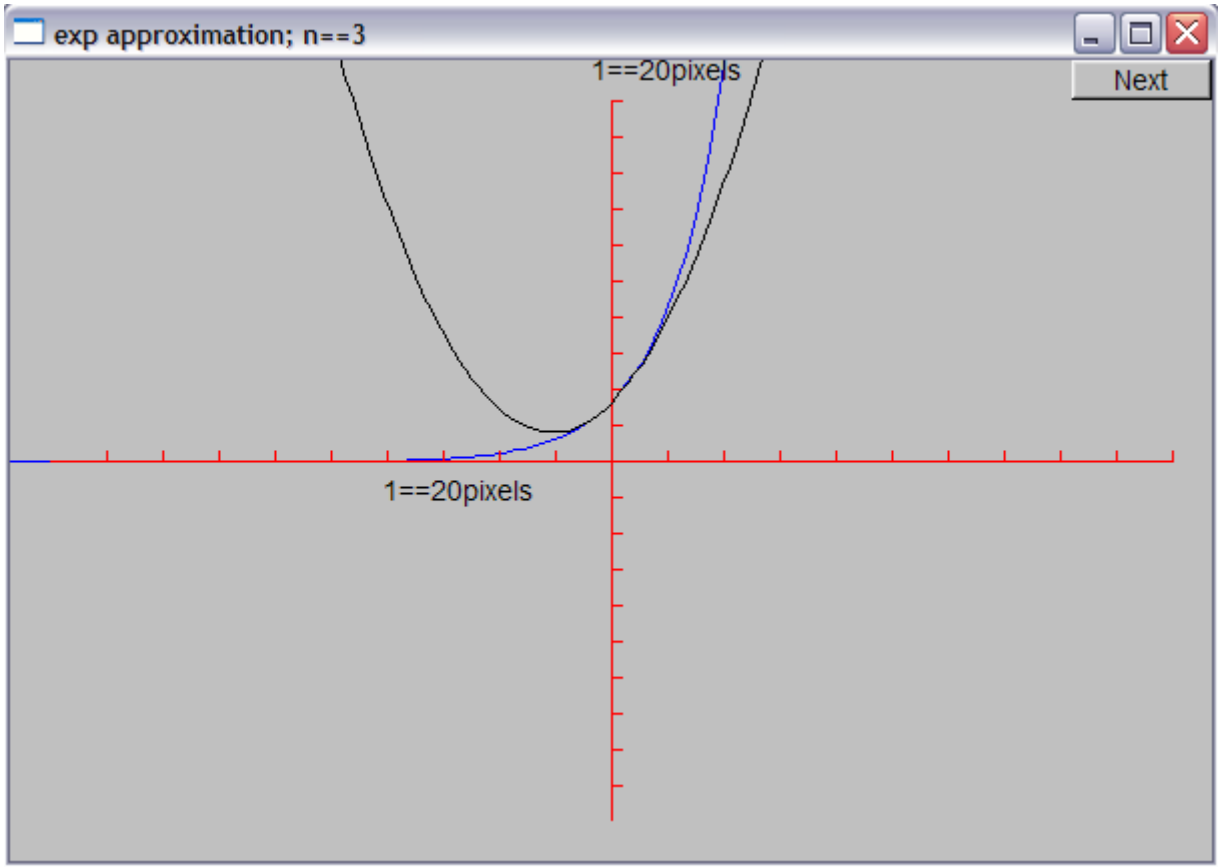
Demo n = 1



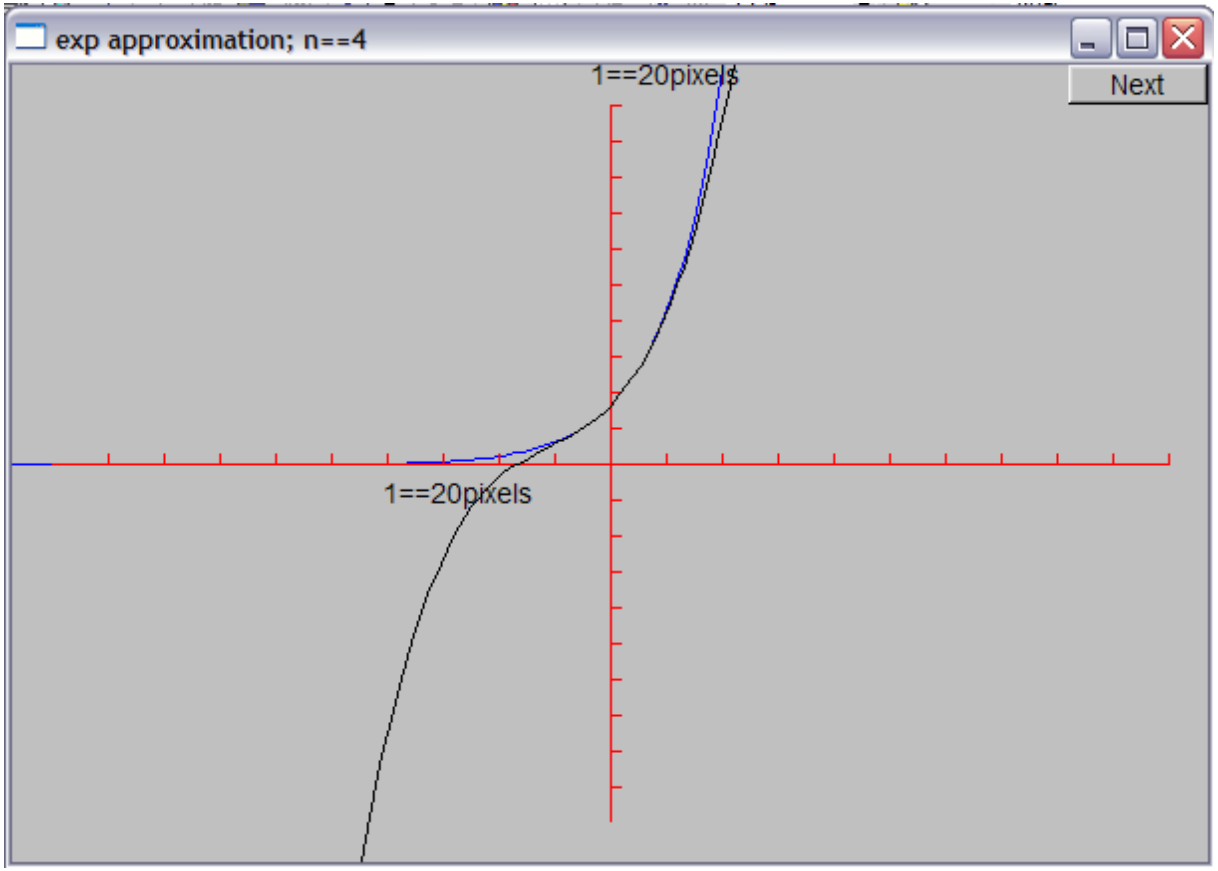
Demo n = 2



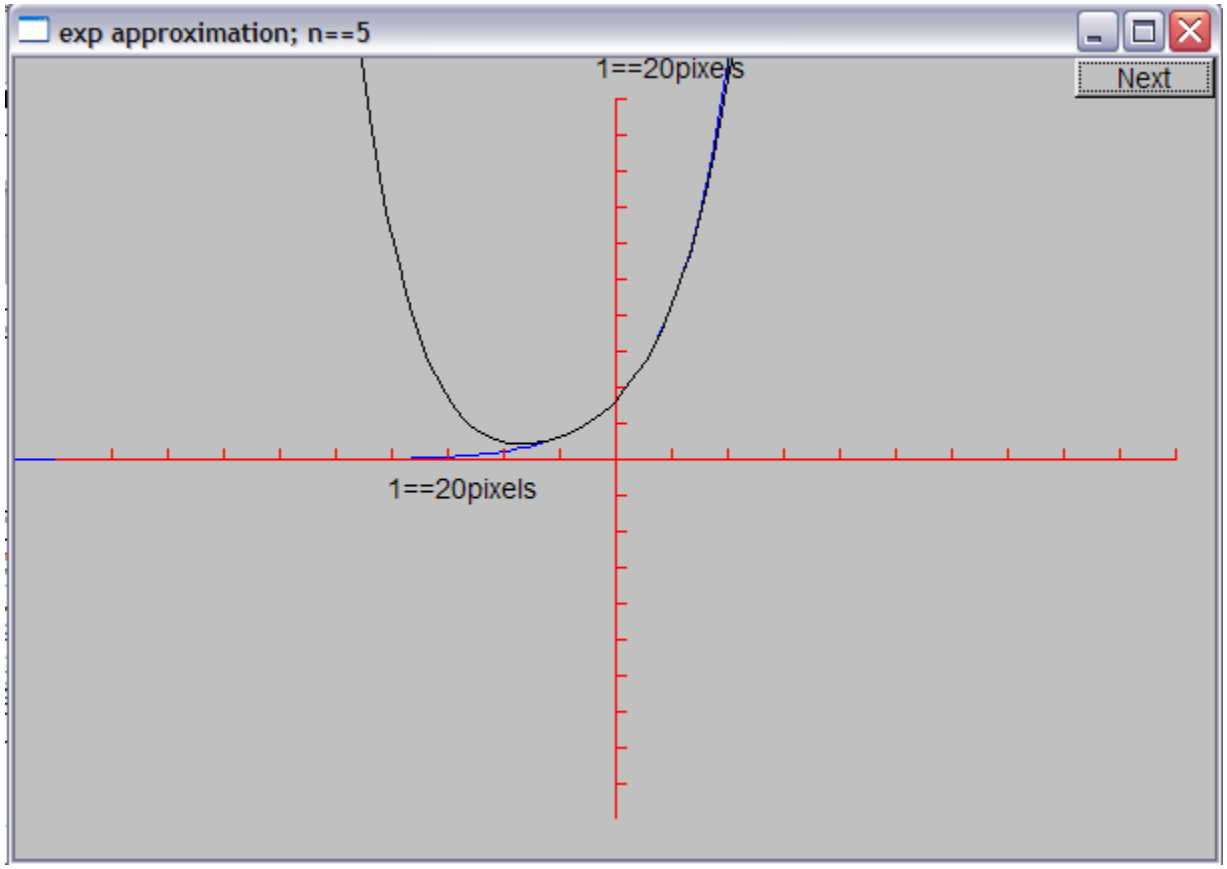
Demo n = 3



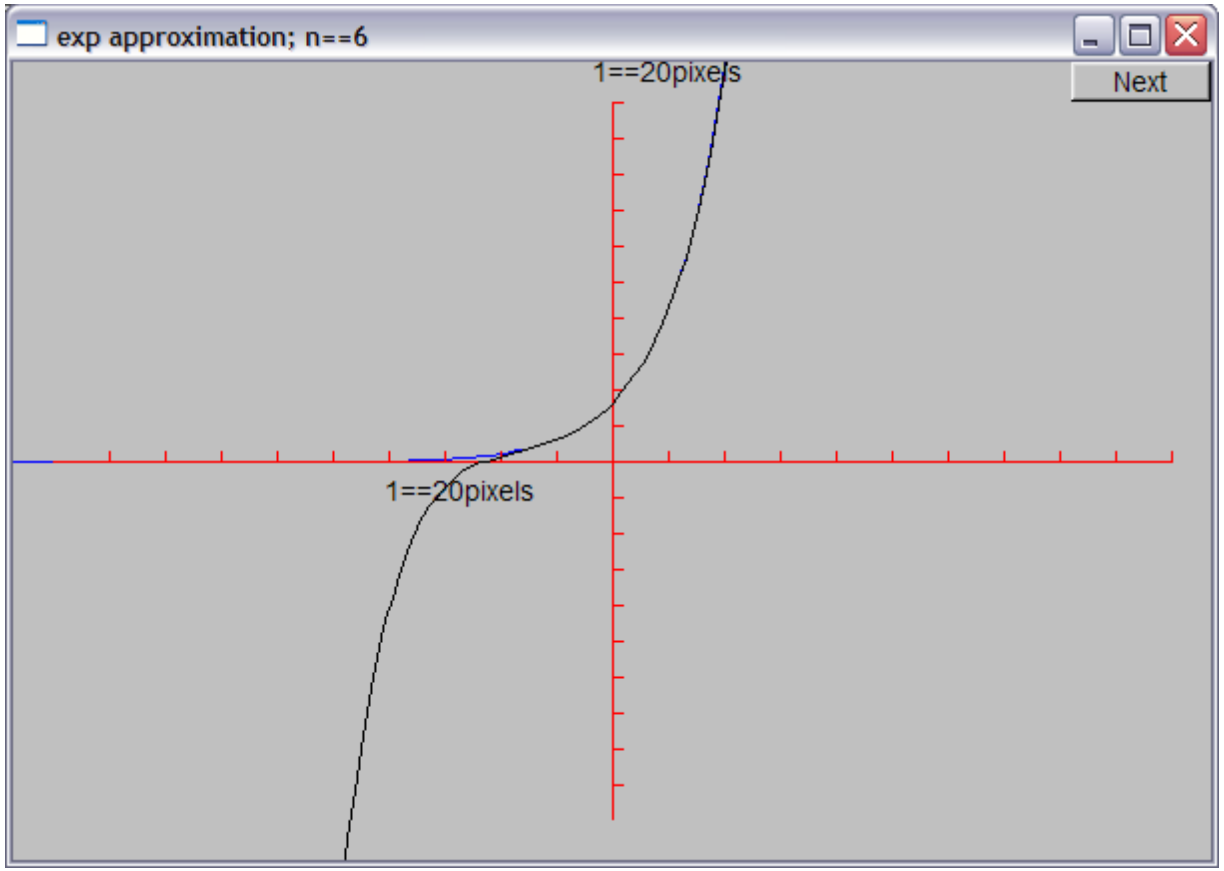
Demo n = 4



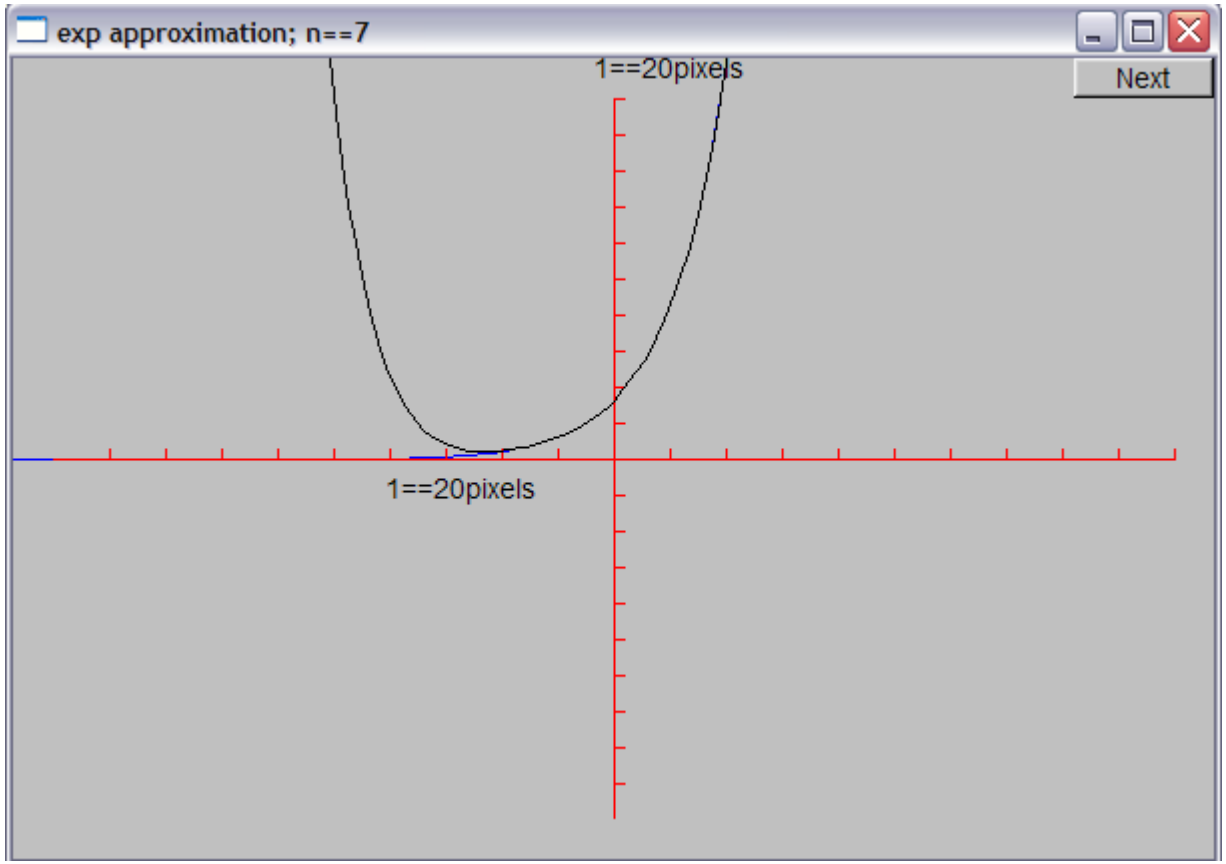
Demo n = 5



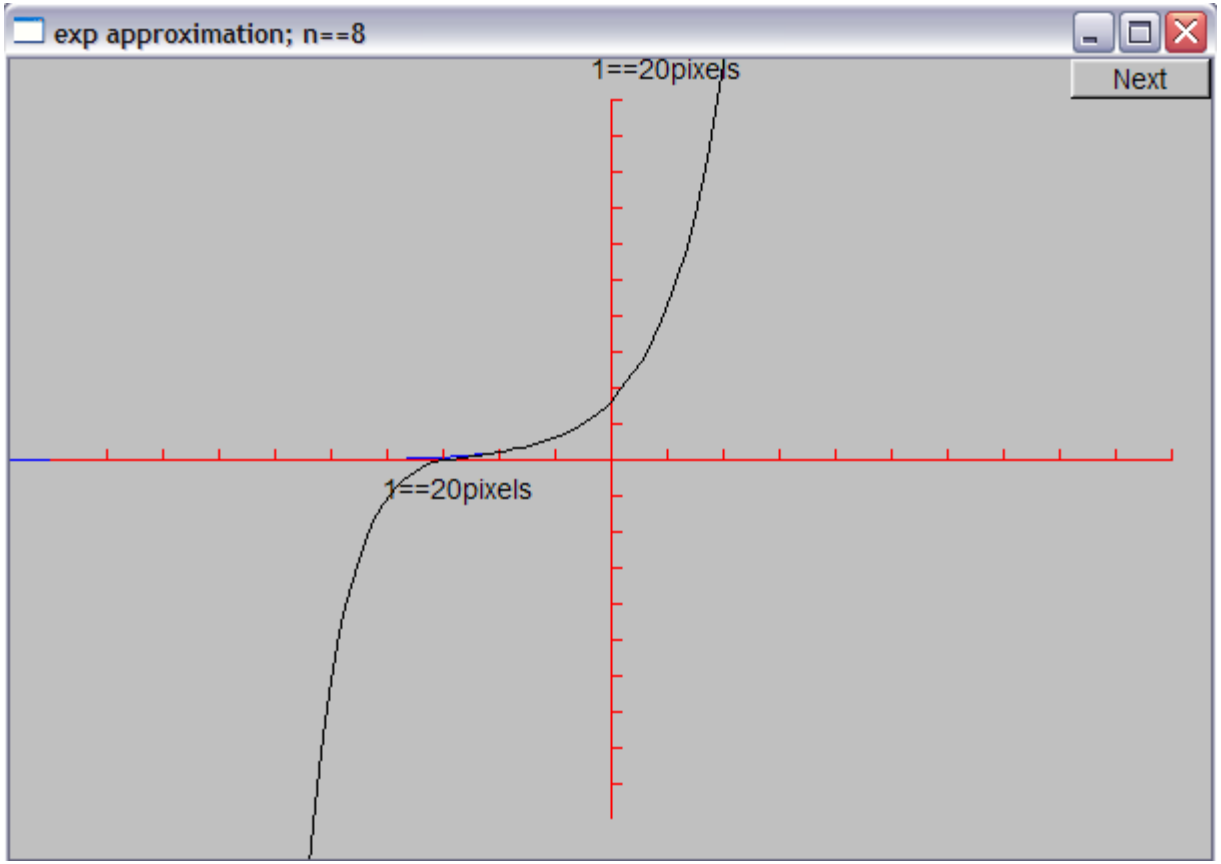
Demo n = 6



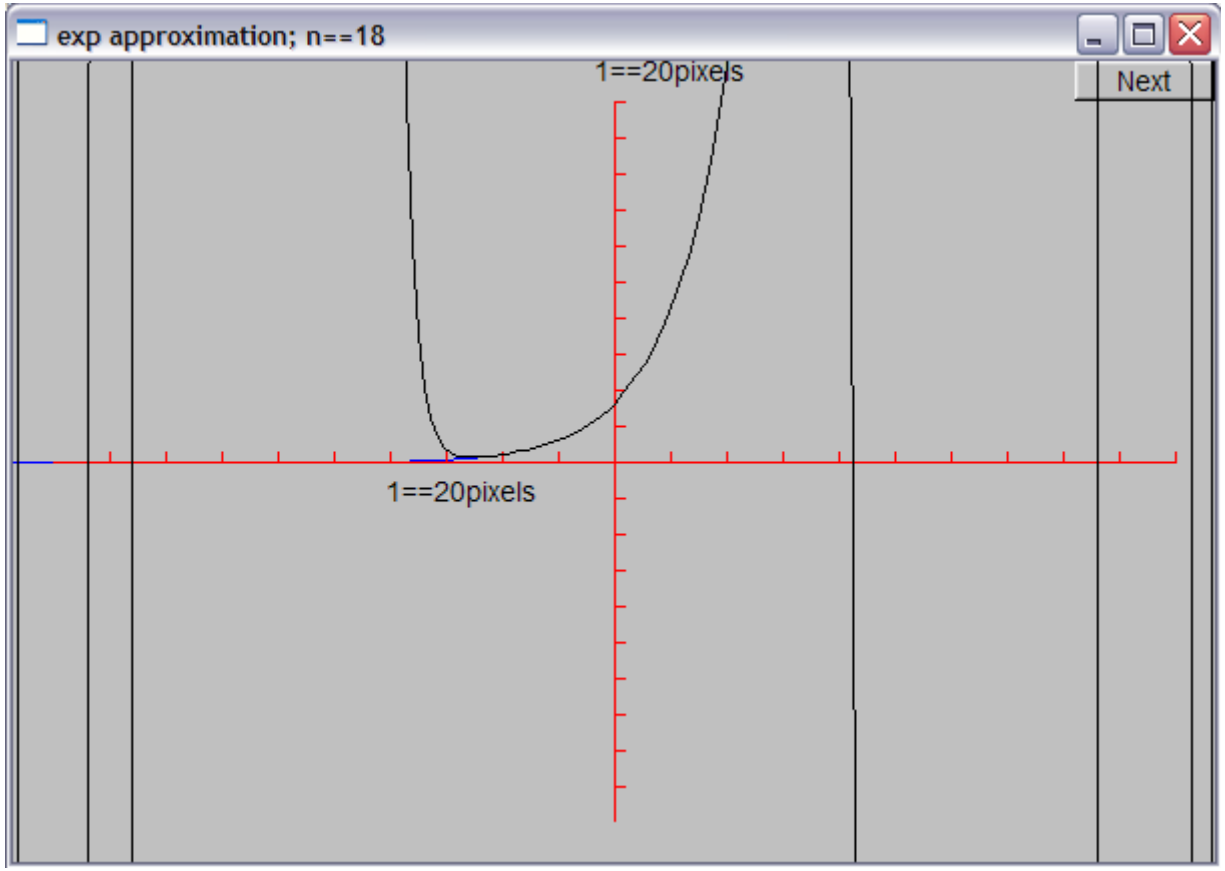
Demo n = 7



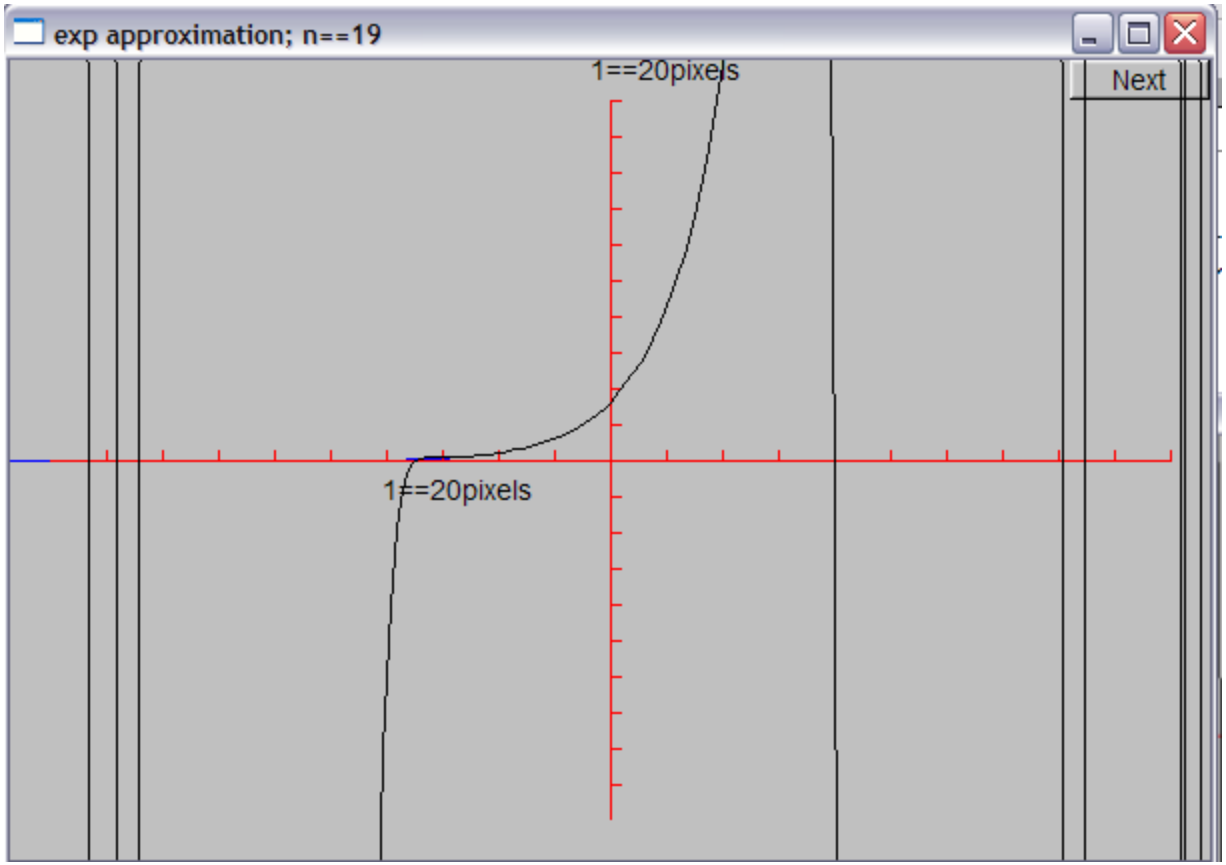
Demo n = 8



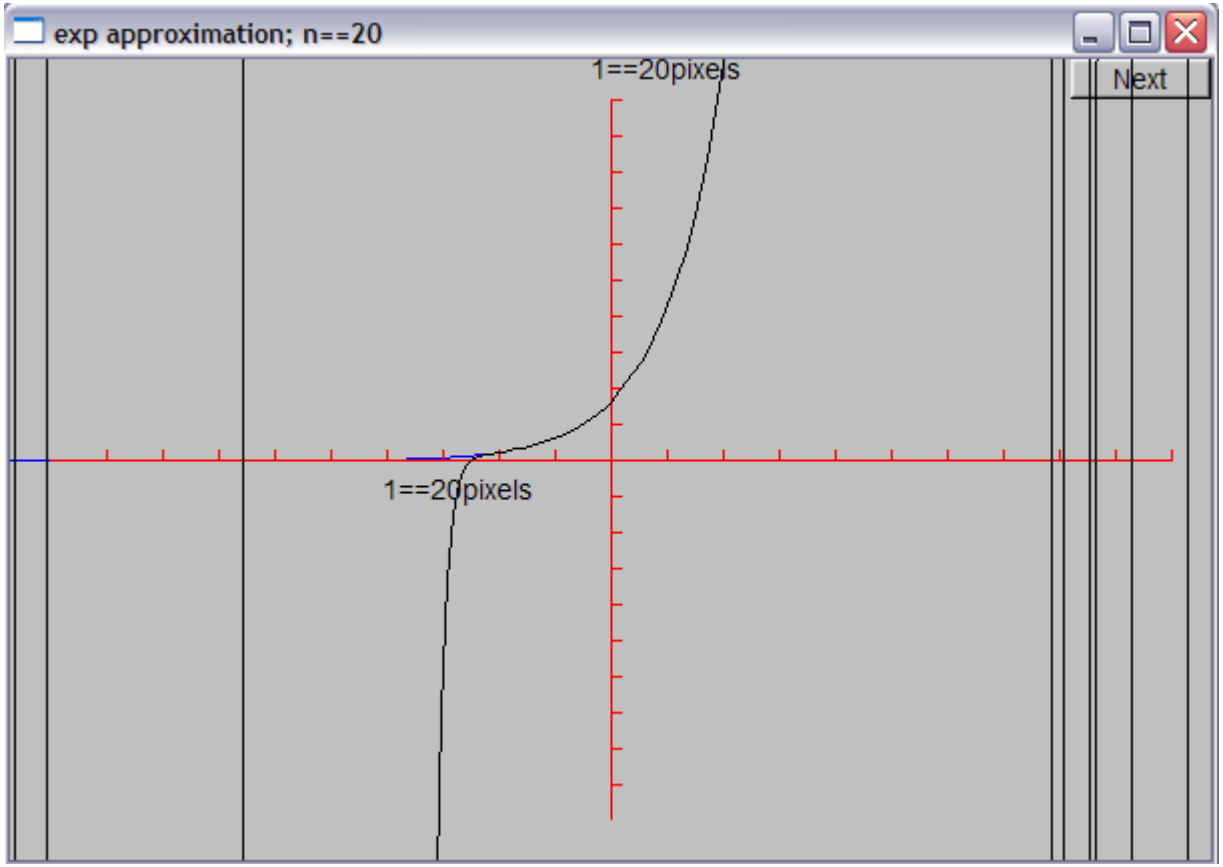
Demo n = 18



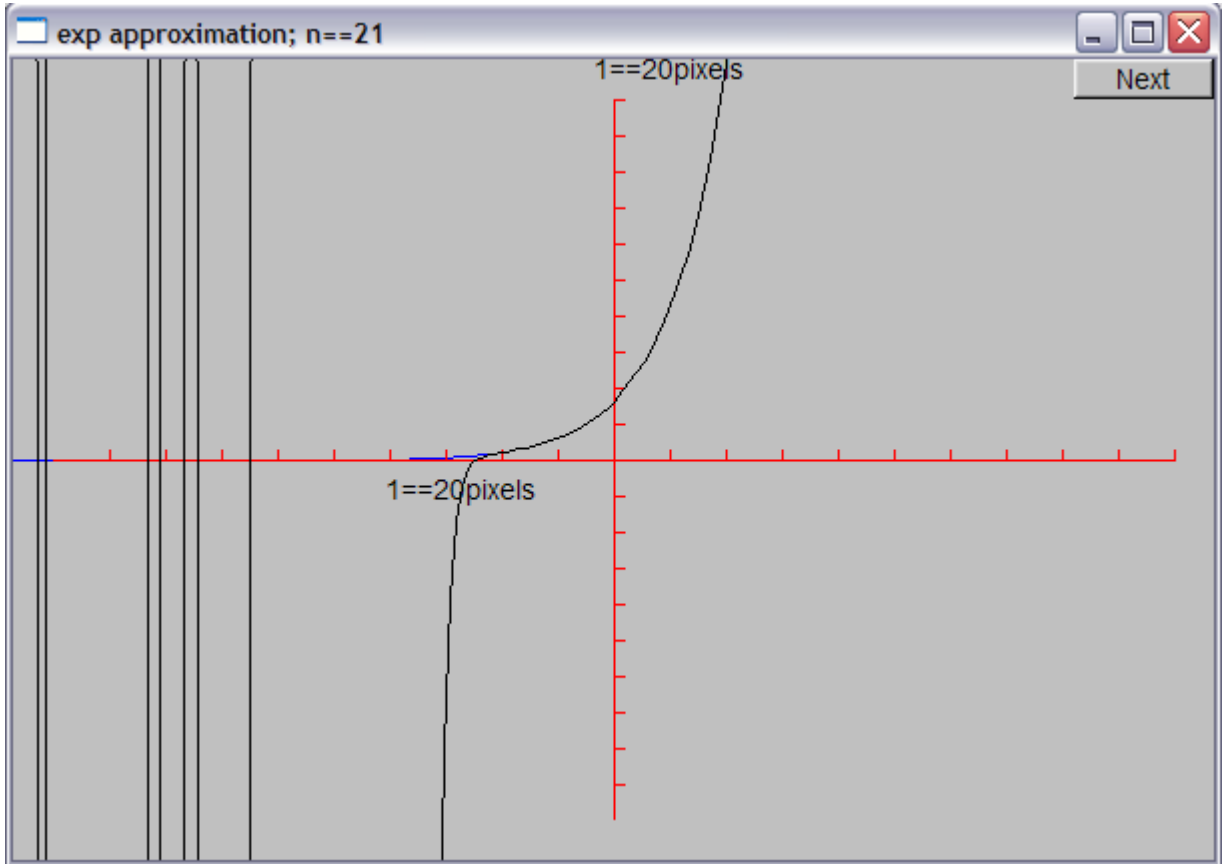
Demo n = 19



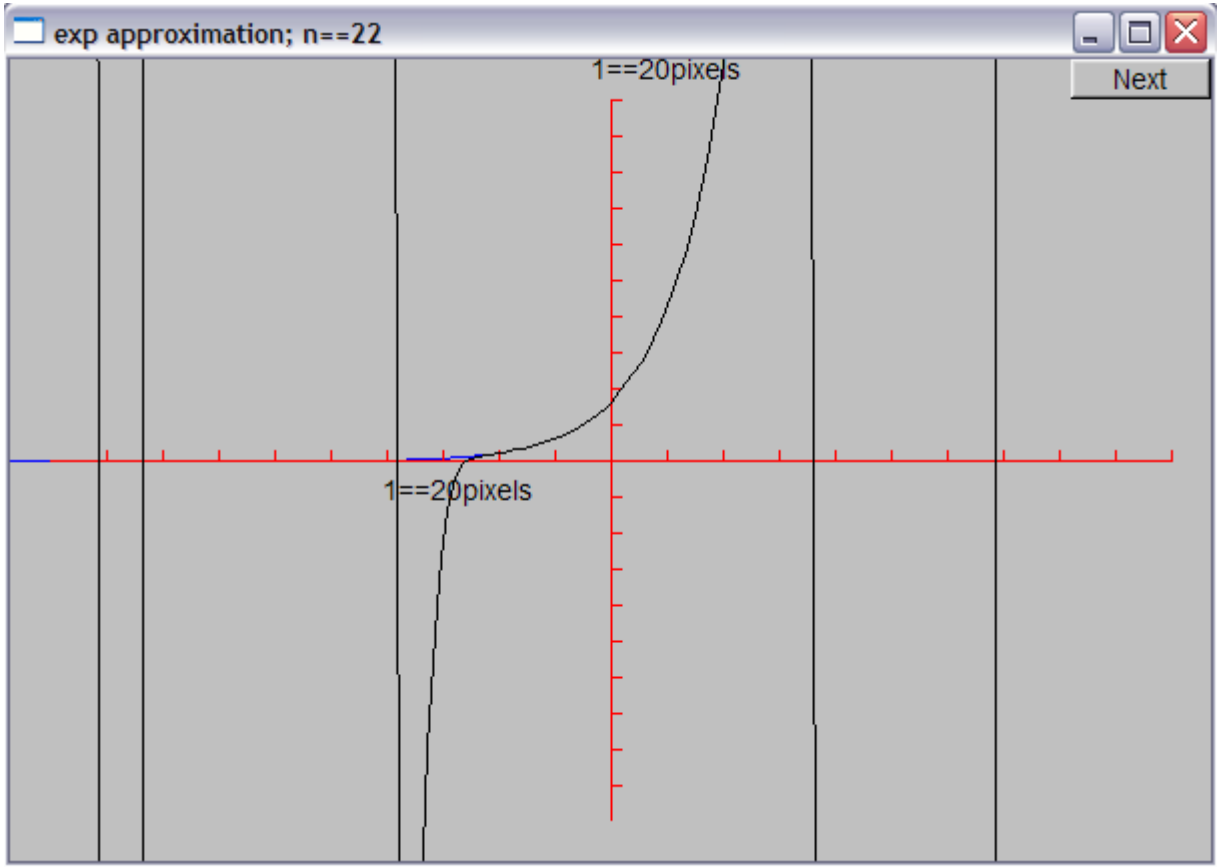
Demo n = 20



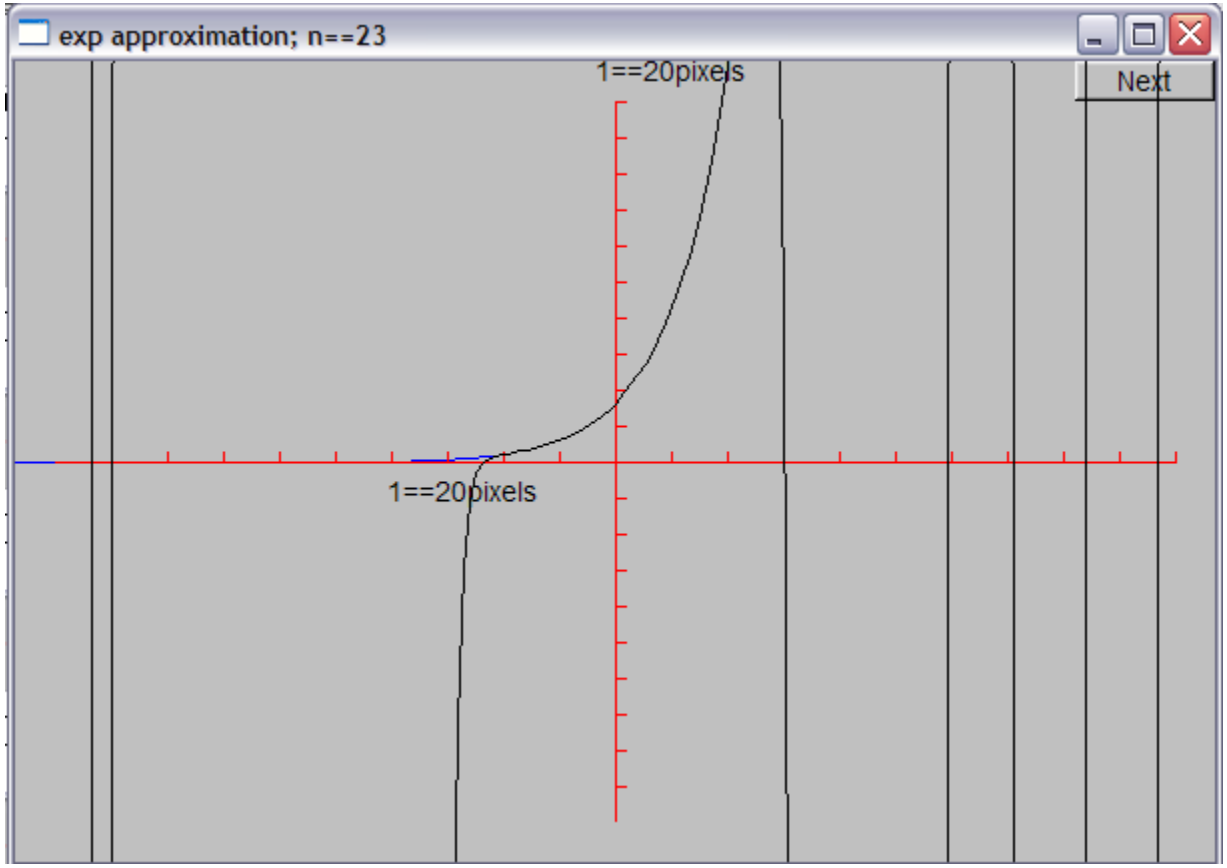
Demo n = 21



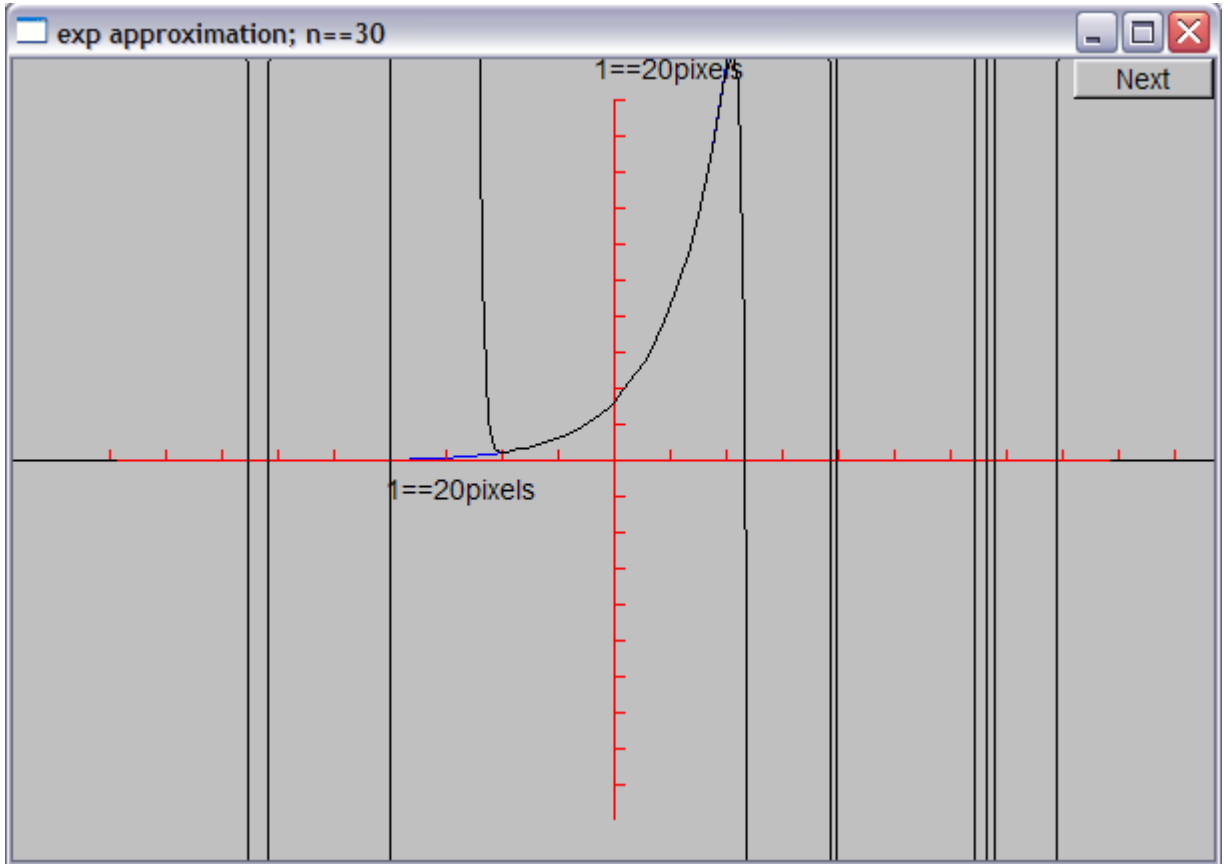
Demo n = 22



Demo n = 23



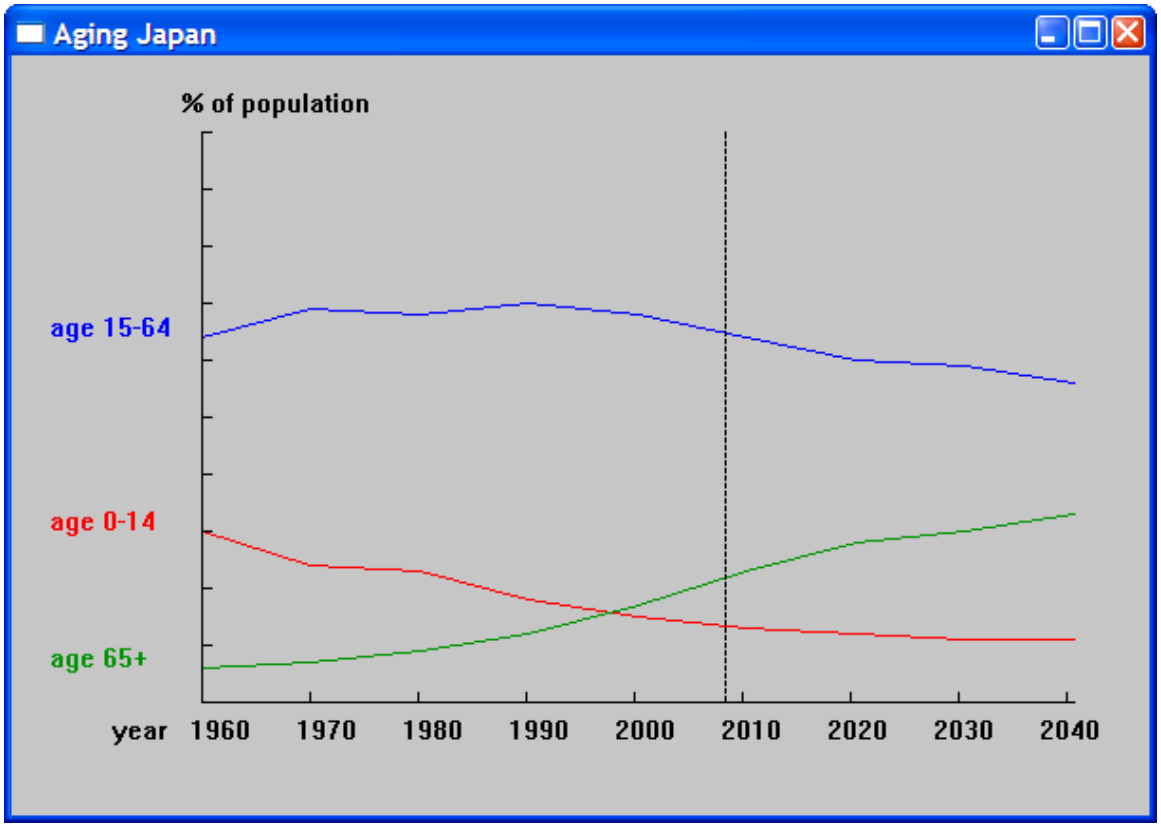
Demo n = 30



为什么图形变得乱七八糟？

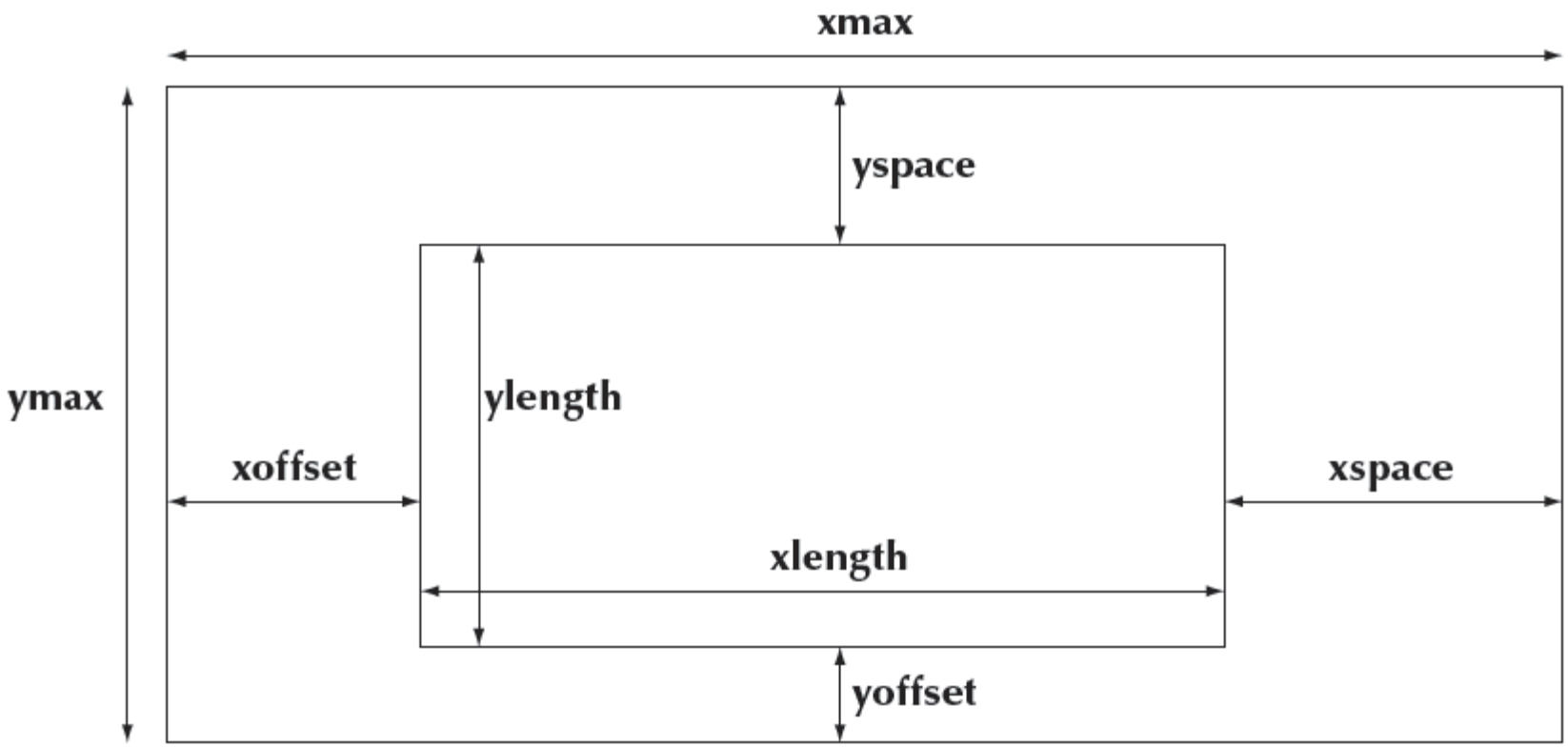
- 浮点数是实数的近似
  - 仅仅是近似
  - 实数值是可以任意大和任意小的
    - 在计算机中，浮点数是固定大小的
  - 有时候，近似值并不能很好地完成你需要的工作
  - 此处，很小的不精确(迭代误差)产生了很大的差错
- 记住
  - 对所有的计算总是要怀疑
  - 检查你的结果
  - 希望你的错误是明显的
    - 你总是应该期望你的代码问题尽早出现 — 在其他人开始使用之前

绘制数据图



- 通常，你需要图形化的是数据，而不是一个定义好的数学函数
  - 上图中，我们使用了三个 **Open_polylines**

绘制数据图



- 小心地设计你的屏幕布局 — 可视化和美观

更多内容参见 15.6 ……

Axis 的代码

```
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_lines() const;
    void move(int dx, int dy);

    void set_color(Color);        // in case we want to change the color of all parts at once

    // line stored in Shape
    // orientation not stored (can be deduced from line)
    Text label;
    Lines notches;
};
```

标签和刻度是单独的对象，允许单独操作

Axis 的代码

```
Axis::Axis(Orientation d, Point xy, int length, int n, string lab)
    :label(Point(0,0),lab)
{
    if (length<0) error("bad axis length");
    switch (d){
    case Axis::x:
    {       Shape::add(xy);                          // axis line begin
            Shape::add(Point(xy.x+length,xy.y));     // axis line end
            if (1<n) {
                    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);        // put label under the line
            break;
    }
    // …
}
```

Axis 实现

```
void Axis::draw_lines() const
{
    Shape::draw_lines();          // the line
    notches.draw();  // the notches may have a different color from the line
    label.draw();                 // the label may have a different color from the line
}

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

void Axis::set_color(Color c)
{
    // ... the obvious three lines ...
}
```

用于提供一致化的风格

Next

- 图形用户接口 GUI
- Windows 和 Widgets
- Buttons 和 Dialog boxes