DM560

Introduction to Programming in C++

Graphing Functions

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

Outline

Overview

- Graphing functions and data
- Scaling

Note

This course is about programming.
Examples – such as graphics – are useful

- to illustrate programming techniques
- to introduce tools for constructing real programs

Hence, observe:

- 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

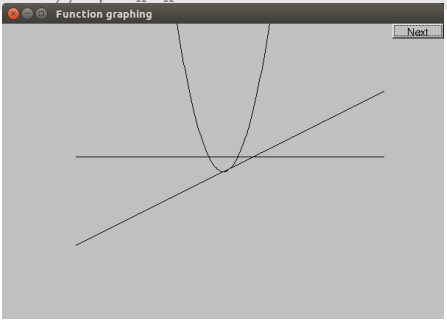
- Start with something really simple (Always remember "Hello, World!")
- We graph functions of one argument yielding one value Plot $(x, f(x))$ for values of x in some range $[r_1, r_2)$
- Let's graph three simple functions:

```
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
```

Graphing Functions

- 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 $[r_1, r_2)$
- Let's graph three simple functions:

```
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?

```
Simple_window win0(Point(100,100),600,400,"Function graphing");

Function s(one,       r_min,r_max,   orig,   n_points,x_scale,y_scale);
Function s2(slope,    r_min,r_max,   orig,   n_points,x_scale,y_scale);
Function s3(square,   r_min,r_max,   orig,   n_points,x_scale,y_scale);

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

win0.wait_for_button( );
```

Stuff to make the graph fit into the window

First point

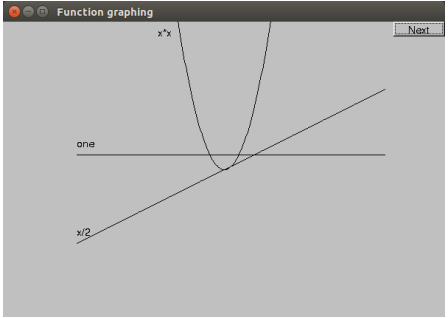
Range in which to graph [x0:xNA)

We Need Some Constants

```
const int xmax = win0.x_max();  // window size (600 by 400)
const int ymax = win0.y_max();

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 = 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
```

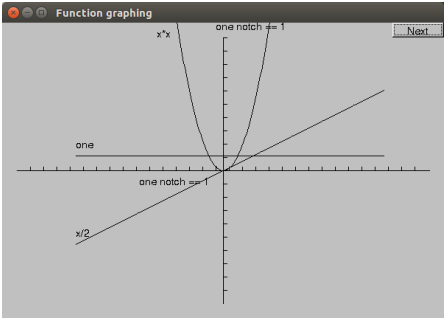
Label the Functions

```
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");
// win0.attach(...)
```



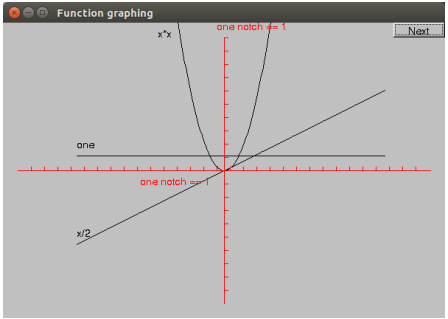
Add x-Axis and y-Axis

```
Axis x(Axis::x, Point(20,y_orig), xmax-40, xmax/x_scale, "one notch == 1 ");
Axis y(Axis::y, Point(x_orig,ymax-20), ymax-40, ymax/y_scale, "one notch== 1");
// win0.attach(...)
```



Set Color

```
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

- `typedef` can be used to declare a new name for a type

```
typedef  int Count;              // now Count means int
```

- We define the type of our desired argument, `Fct`

```
typedef double Fct(double);      // now Fct means  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 x/2; }   // y==x/2
double square(double x) { return x*x; } // y==x*x
```

Now Define 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 Cartesian (0,0)
            Count 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   // relative to orig (why minus?)
           );
};
```

Implementation of Function

```
Function::Function(Fct f,
                   double r1, double r2,        // range
                   Point xy,
                   Count 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! They can generate confusion and errors.

- Provide defaults for some arguments
- Arguments with deafults must be the trailing ones
- Specify deafult values in the header files
- Default arguments are often useful for constructors

```
struct Function : Shape {
   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"? How can we improve it?

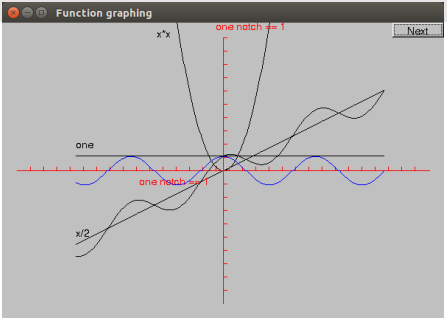
- we could create classes to handle all those position and scaling arguments. For example a class `scale` containing all elements to scale.
- if you can't do something genuinely clever, do something simple, so that the user can do anything needed (Such as adding parameters so that the caller can control precision)

Some More Function

```
#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);
```

Some More Function

```
#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);
```



Standard Mathematical Functions (1/2)

made available by including < 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);
```

Standard Mathematical Functions (2/2)

made available by including < 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 y, double x);       // atan(y/x)
double fmod(double d, double m);        // floating-point remainder
double ldexp(double d, int i);  // d*pow(2,i)
```

Why Graphing?

Visualization

- Is key to understanding in many fields
  (how could you understand a sine curve if couldn't ever see one?)
- Is used in most research and industry:
  Science, medicine, business, telecommunications, control of large systems
- Can communicate large amounts of data simply

An Example

Taylor expansion of e^x about $x = 0$

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \frac{x^5}{5} + \frac{x^6}{6} + \frac{x^7}{7} + \ldots$$

Implementation:

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

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

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

Example: "Animate" Approximations to e^x

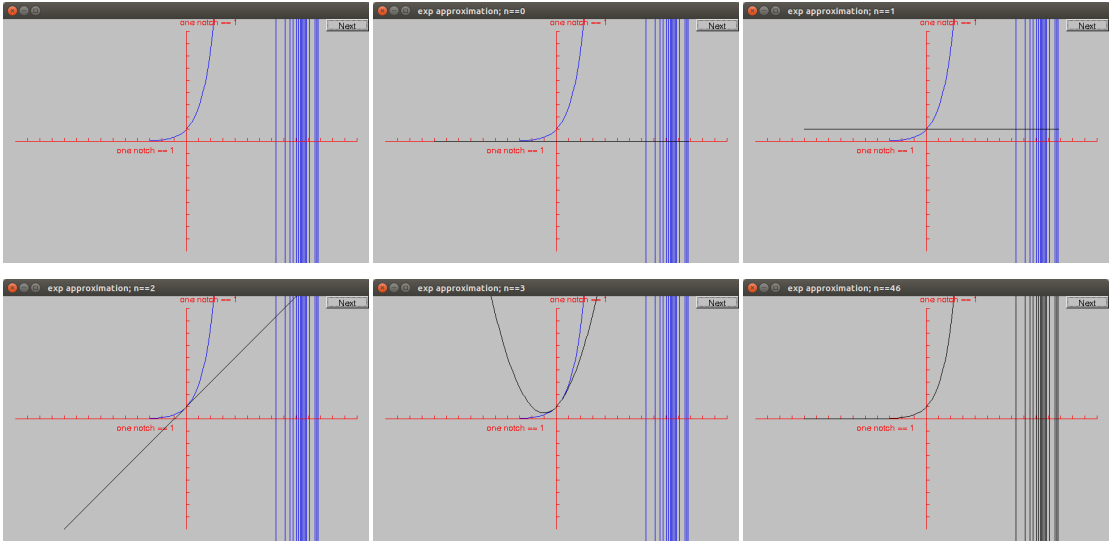
```
double expN(double x)   // sum of expN_number_of_terms terms of x
{
        return expe(x,expN_number_of_terms);
}
```

```
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);
```

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

Example: Result



Why Did the Graph Go so Wild?

Floating-point numbers are an approximations of real numbers

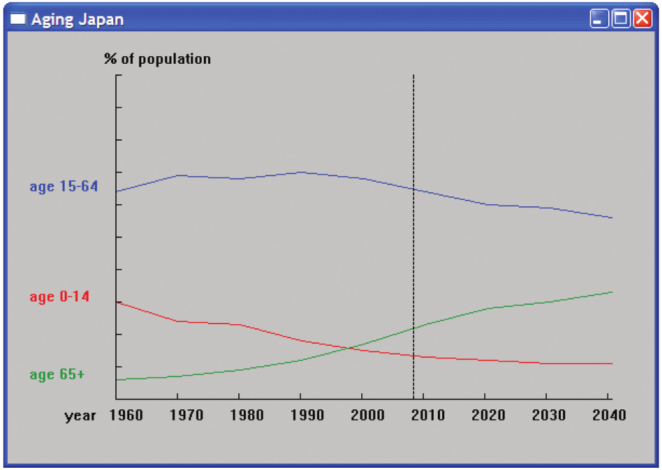
- Real numbers can be arbitrarily large and arbitrarily small
  Floating-point numbers are of a fixed size and can't hold all real numbers
- Sometimes the approximation is not good enough for what you do
- Small inaccuracies (rounding errors) can build up into huge errors

Hence, always

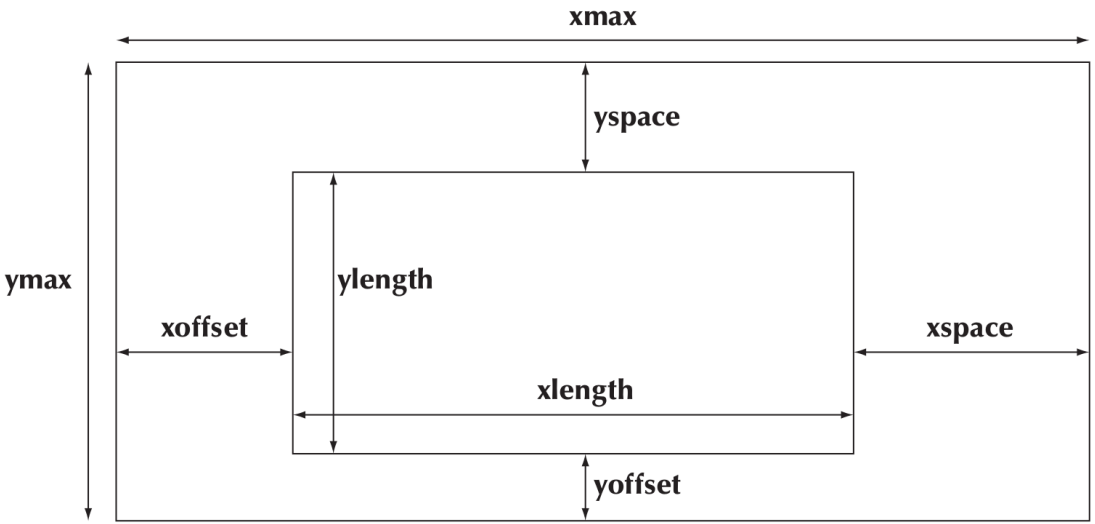
- be suspicious about calculations
- check your results
- hope that your errors are obvious: You want your code to break early – before anyone else gets to use it

Graphing Data

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



Screen Layout



Code for Axis: Declaration

```
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 color all parts at once

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

Code for Axis: Implementation

```
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;
   }
   // ...
}
```

Code for Axis: Implementation

```
void Axis::draw_lines() const
{
   Shape::draw_lines(); // the line
   notches.draw_lines(); // 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 ...
}
```