图形库使用参考

一 简介

※本课程的图形库是一个简单的图形用户界面库

⊙实现简单的图形输出和图形用户界面

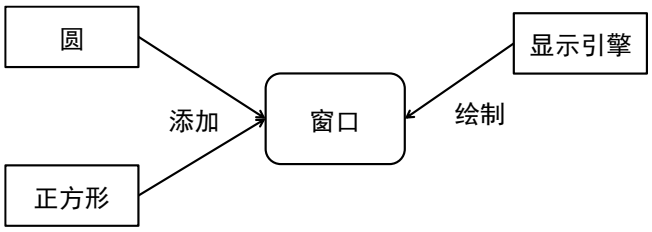
⊙展示面向对象编程的概念及其语言特征

⊙帮助理解其他GUI库

※显示模型

⊙利用图形系统提供的基本对象（如线、矩形、文本等）组合出更复杂的对象

⊙将图形对象添加到一个表示物理屏幕的窗口

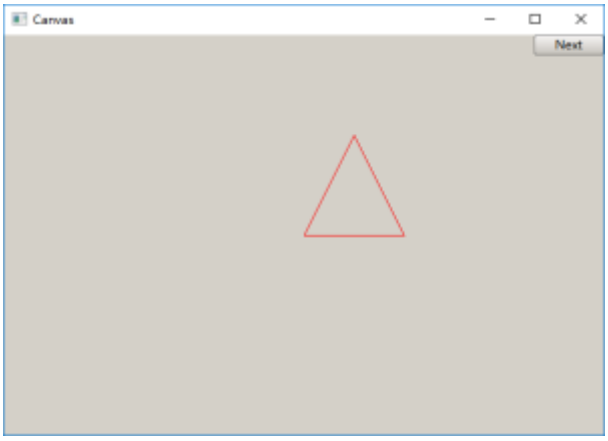


⊙一个显示引擎负责将窗口中的图形对象显示出来

※一个简单例子（chapter.12.3.cpp）

```
// 图形库的头文件：
#include "Simple_window.h"          // 用来绘制图形的简单窗口
#include "Graph.h"                  // 二维图形
int main(){
    using namespace Graph_lib; // 图形库的名字空间
    Point tl = {100,200};       // 一个点，作为窗口的左上角
    Simple_window win = {tl,600,400,"Canvas"};// 创建一个简单窗口
    Polygon poly;               // 创建一个图形对象（多边形）
    poly.add(Point{300,200});       // 添加 3 个顶点
    poly.add(Point{350,100});
    poly.add(Point{400,200});
    poly.set_color(Colors::red);   // 设置多边形的颜色
    win.attach(poly);               // 将多边形添加到窗口上
    // 将控制权交给显示引擎，按下“Next”按钮后程序结束
    win.wait_for_button();
}
```

⊙输出结果



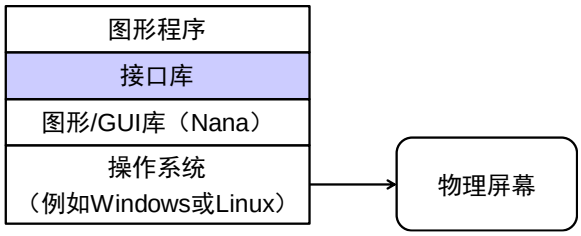
※说明：

本课程的图形库基于《C++程序设计原理与实践》一书（以下简称为PPPC）提供的图

形库修改而成。本文简单介绍利用图形库进行绘图和GUI编程的方法，以及图形库与PPPC原始版本的区别。更详细的说明可见PPPC第12至第16章的内容。

1.1 图形库结构

C++没有标准的GUI库，在这里我们使用的GUI工具包名为Nana，具体可参考nanapro.org网站并下载相应的工具包：nana1.7.4.zip。本课程在Nana的基础上提供了一个简单的接口库，约1300行代码，20多个类，简单易用，隐藏了下层GUI库的复杂细节。



1.2 图形库安装使用(Visual C++)

由于图形接口库中广泛使用了C++11特性，需要使用新版本的Visual C++编译器（2019）。

(1) 编译Nana库

⊙解压源码包 nana1.7.4.zip

⊙进入nana/build/vc2019子目录

⊙打开解决方案文件nana.sln

⊙生成解决方案

(2) 解压图形接口库GUI_v3.zip

(3) 创建图形程序项目

⊙控制台项目

⊙在向导中选中“空项目”

(4) 加入图形接口库源文件

⊙ 头文件 ：

Point.h	点
Graph.h	图形（线段、矩形等）
GUI.h	GUI组件（按钮、输入框等）
Simple_window.h	简单窗口
Window.h	窗口
Drawing.h	绘图对象

⊙代码文件 ：

```
Graph.cpp
GUI.cpp
Window.cpp
Drawing.cpp
```

(5) 设置图形程序项目属性

假设Nana在“D:\nana”目录下，图形接口库和程序项目处在并列的目录中。

⊙附加包含目录（配置属性 -> C/C++ -> 常规）

```
..\GUI_v3;D:\nana\include
```

⊙ 附加库目录（配置属性 -> 链接器 -> 常规）

```
D:\nana\build\bin
```

⊙ 附加依赖项（配置属性 -> 链接器 -> 输入）

```
nana_$(PlatformToolset)_$(Configuration)_$(PlatformShortName).lib
```

从父级或项目默认设置继承

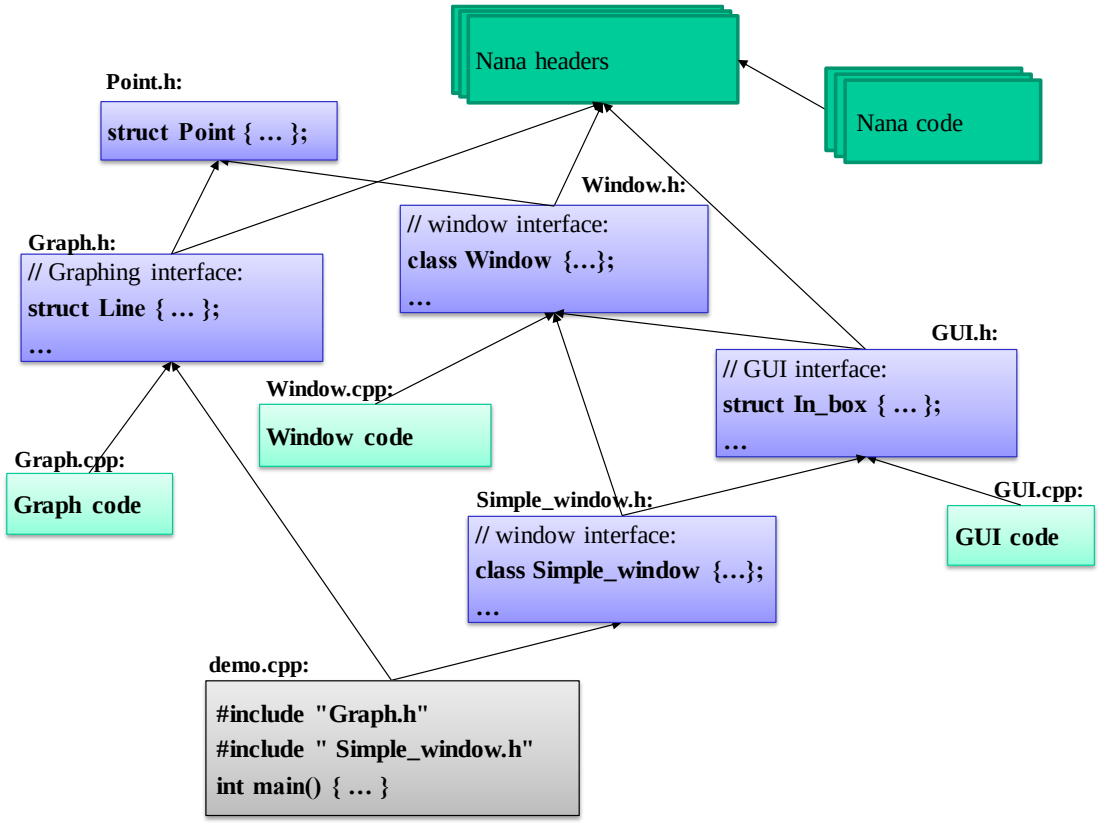
图形程序项目的编译选项，包括配置、平台、运行库（配置属性 -> C/C++ -> 代码生成）等应同编译nana时一致。

(6) 编写图形程序

新建一个或多个源文件（.cpp, .h）

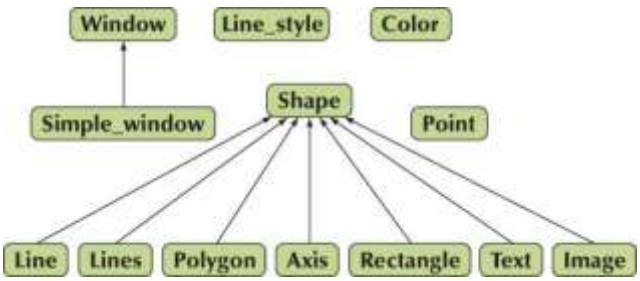
(7) 编译、运行

1.3 代码结构



二 图形类

在 Graph.h 中定义了十多个图形类，如下图所示：



在这里可以看到，Shape 类是其他图形类的基类。

2.1 点和线

二维图形由点和线组成。点用 Point{x,y}表示，其中，x,y 是一对整数值，分别代表了窗口画面上坐标值。（在这里，屏幕的左上角设定 x=0，y=0）。

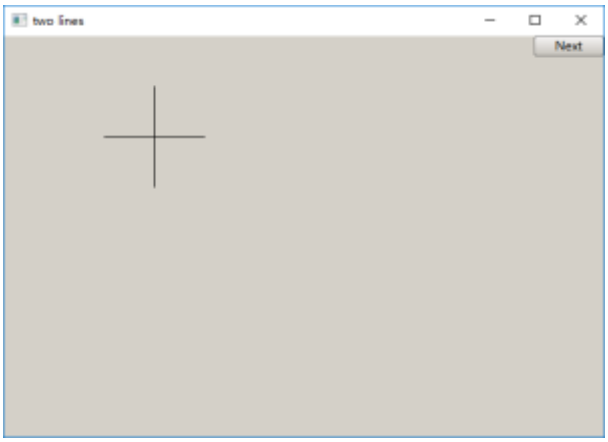
```
struct Point {
    int x, y;
};
```

※一对点确定一条线段：即Line由两个Point定义

```
struct Line : Shape {
    Line(Point p1, Point p2);
};
```

⊙绘图代码实例（chapter.13.2.cpp）

```
// draw two lines
Simple_window win1 = {{100,100}, 600,400, "two lines"};
// make a horizontal line
Line horizontal = {{100,100}, {200,100}};
// make a vertical line
Line vertical = {{150,50}, {150,150}};

win1.attach(horizontal);     // attach the lines to the window
win1.attach(vertical);
win1.wait_for_button();     // display!
```



※Lines将多条线段组合起来，成为一个整体，以便进行统一操作

```
struct Lines : Shape {           // 一组相关的线
       Lines() {}                // empty
       // initialize from a list of Points
       Lines(std::initializer_list<Point> lst);
       // add a line defined by two points
       void add(Point p1, Point p2);
};
```

一个 Lines 对象是多个 Line 的组合，其中每一条 Line 由一对 Points 组成，对于上一个例子中的两条 Line 作为一个图形对象的话，可以用 Lines 作如下定义：

```
Lines x;
x.add({100,100}, {200,100});   // first line: horizontal
x.add({150,50}, {150,150});    // second line: vertical
```

对于上述对象也可以用以下代码实现：

```
Lines x = {{100,100}, {200,100}, {150,50}, {150,150}};
```

2.2 颜色和线型

※Color是用于表示颜色和透明性的一个类

```
using Colors = nana::colors;  // 定义了各种颜色的名称
struct Color {
    enum Transparency { invisible = 0, visible = 255 };

    Color(int rr, int gg, int bb, int vv= visible);
    Color(Colors cc, Transparency vv = visible);
    Color(Transparency vv) : r{0}, g{0}, b{0}, v(vv) { }

    int red()   const { return r; }
    int green() const { return g; }
    int blue()  const { return b; }
    char visibility() const { return v; }
    void set_visibility(Transparency vv) { v = vv; }

private:
    unsigned char r, g, b, v;
};
```

例：

```
    x.set_color(Colors::red);   //  将上例中的两条线同时设置为红色
```

※Line_style定义线的类型和宽度

在一个窗口中绘制 Lines 时，可以通过颜色、线型或者将两者结合将它们区分开来。线型是用来描述线的外形的一种模式，使用方法如下：

例：

```
    x.set_style(Line_style::dot);   //  将上例中的两条线为点状线而非实线
```

※颜色和线型的综合实例见chapter.13.5-1.cpp。

注：Nana 不支持线型和宽度。程序中定义的线型和宽度对显示没有影响。

2.3 折线和多边形

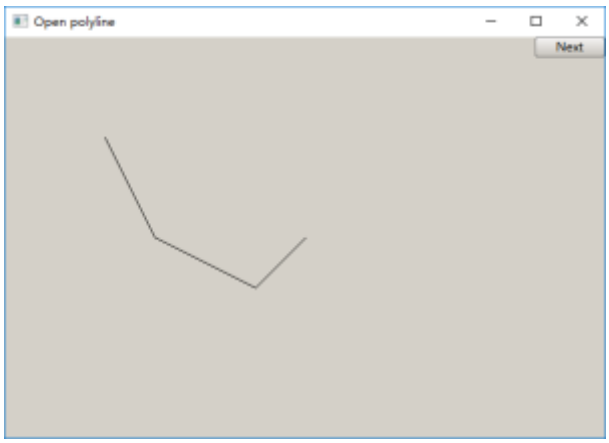
※Open_polyline(非闭合折线)是由依次相连的线段序列组成的形状，由一个点的序列定义。

Open_polyline 类的定义如下：

```
    struct Open_polyline : Shape {       // 非闭合的线段序列
        Open_polyline();
        Open_polyline(std::initializer_list<Point> lst);
        void add(Point p) { Shape::add(p); }
    };
```

如下代码（chapter.13.6.cpp）：

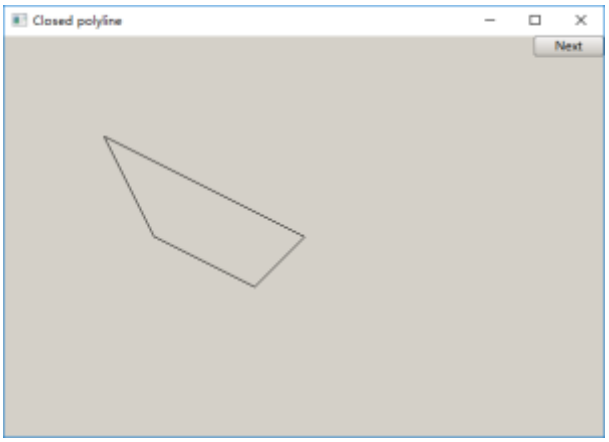
```
Open_polyline opl = {{100,100}, {150,200}, {250,250}, {300,200}};
```
可以得到如图所示的由 4 个点所连结的图像（一条折线）。



※Closed_polyline（闭合折线）和Open_polyline很相似，只是多一条从最后一点到第一点的闭合线

如将上例中的 Open_polyline 改为 Closed_polyline;

```
Closed_polyline cpl = {{100,100}, {150,200}, {250,250}, {300,200}};
```
结果如下（chapter.13.7.cpp）：



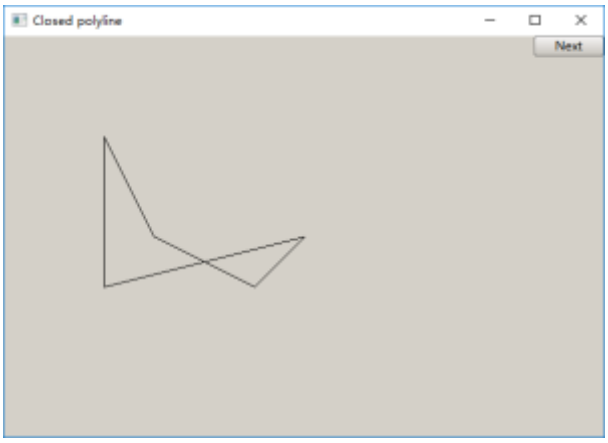
Closed_polyline 类的定义如下：

```
struct Closed_polyline : Open_polyline {   // 闭合的线段序列
    // 继承 Open_polyline 的构造函数
    using Open_polyline::Open_polyline;
    void draw_lines(Graphics& g) const;
};

void Closed_polyline::draw_lines(Graphics& g) const {
    // 先绘制“开放部分”
    Open_polyline::draw_lines(g);
    // 再绘制闭合线
    if (2<number_of_points() && color().visibility())
      g.line(point(number_of_points()-1), point(0));
}
```

※Polygon和Closed _polyline很相似，唯一的区别是Polygon不允许出现交叉的线。

Polygon在加入点时进行有效性检查。前面定义的cpl是一个多边形，但再加入一个点后就不是了（chapter.13.8-1.cpp）：

```
cpl.add({100,250});
```



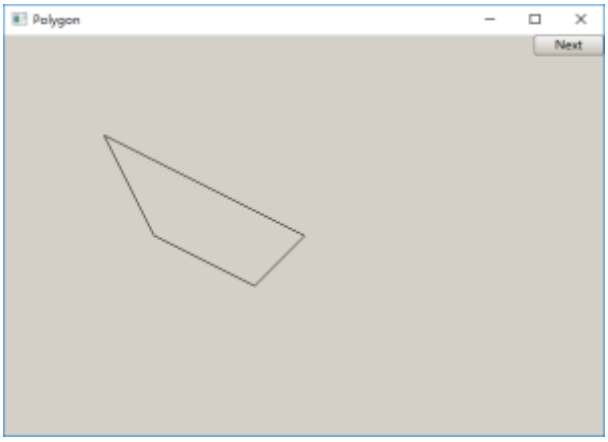
基于此，可以定义 Polygon 如下：

```
struct Polygon : Closed_polyline {     // 不交叉的闭合线段序列
    // 继承 Closed_polyline 的构造函数
    using Closed_polyline::Closed_polyline;
    void add(Point p);
    void draw_lines(Graphics& g) const;
};

void Polygon::add(Point p) {
    // 检查新加的线不同已有的线交叉
    // (代码略)
    Closed_polyline::add(p);
}
```

可以用初始化列表建立一个 Polygon，代码如下（chapter.13.8-2.cpp）：

```
Polygon poly = {{100,100}, {150,200}, {250,250}, {300,200}};
```

其图形为：



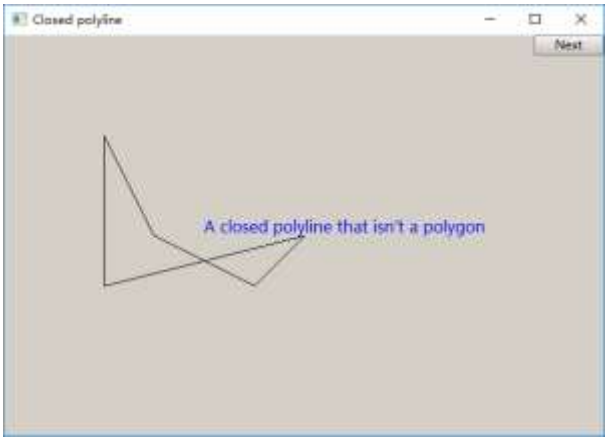
2.4 文本

※Text类用来在窗口的指定位置显示文本。

例（chapter.13.11.cpp）：

```
Text t = {{200,180}, "A closed polyline that isn't a polygon"};
t.set_color(Colors::blue);
```

可以得到图像：



在这里，一个 Text 对象定义了起始位置为(200,180)的一行文本，其中的起始位置为文本行的左上角。不要在字符串中放入换行符，图形库不能显示多行文本。Text 的定义如下：

```
struct Text : Shape {
    Text(Point x, const string& s) : lab{s} { add(x); }

      void draw_lines(Graphics& g) const;

      void set_label(const string& s) { lab = s; }
      string label() const { return lab; }
      void set_font(Font f) { fnt = f; }
      Font font() const { return fnt; }
private:
      string lab;      // 文本内容
      Font fnt;
};
```

2.5 字体

※通过Font类可以为文本设置字体和字号。

例：

```
// Times 字体，加粗，20 磅
t.set_font({"Times", true, false, 20});
```

Font 的定义如下：

```
struct Font {
    Font(const string& name, unsigned size)
     : name_{name}, size_{size}, bold_{false}, italic_{false} {}
```

```
    Font(const string& name = "", bool bold = false, bool italic = false,
          unsigned size = 12)
    : name_{name}, size_{size}, bold_{bold}, italic_{italic} {}

    string   name() const { return name_; }
    unsigned size() const { return size_; }
    bool     bold() const { return bold_; }
    bool   italic() const { return italic_; }

private:
    string name_;
    unsigned size_;
    bool bold_, italic_;
};
```

2.6 矩形

※矩形是屏幕上最常见的形状，因此GUI系统直接支持矩形。

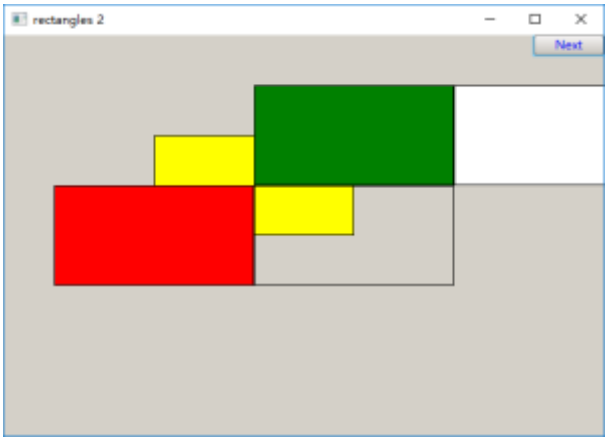
```
    struct Rectangle : Shape {
        // 使用一个顶点（左上角）和宽度、高度定义
        Rectangle(Point xy, int ww, int hh);
        // 使用两个顶点（左上角和右下角）定义
        Rectangle(Point x, Point y);
        void draw_lines(Graphics& g) const;

        int height() const; { return h; }     // 高度
        int width() const;  { return w; }     // 宽度

    private:
        int h, w;
    };
```

例：应用上面的定义在屏幕上画了 5 个矩形，代码设计如下（chapter.13.9-4.cpp）：

```
    Rectangle rect00 = {{150,100},200,100};
    Rectangle rect11 = {{50,50}, {250,150}};    // 在 rect11 下方
    Rectangle rect12 = {{50,150}, {250,250}};   // 在 rect11 右侧
    Rectangle rect21 = {{250,50},200,100};      // 在 rect21 左侧
    Rectangle rect22 = {{250,150},200,100};

    rect00.set_fill_color(Colors::yellow);
    rect11.set_fill_color(Colors::blue);
    rect12.set_fill_color(Colors::red);
    rect21.set_fill_color(Colors::green);
```

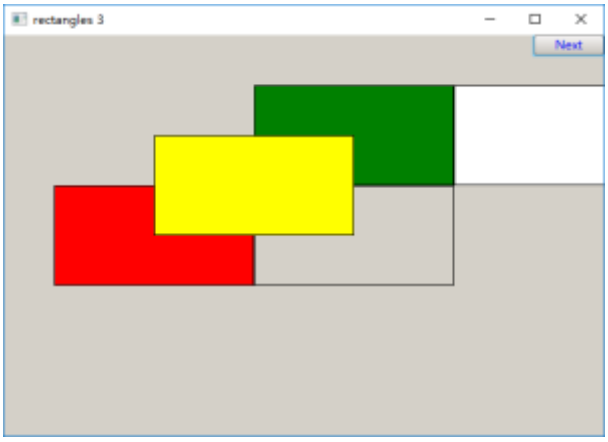
可以用代码将图形在窗口内移动，如：

```
rect11.move(400,0);     // 移到 rect21 右侧
rect11.set_fill_color(Colors::white);
win12.set_label("rectangles 2");
```



此外，窗口中的图形是有层次的，Window 类提供了一种重排图形层次的简单方法：Window::put_on_top())函数可以通知窗口将一个对象放在在最顶层。如：

```
win12.put_on_top(rect00);
win12.set_label("rectangles 3");
```



注意，矩形除了可以被填充某种颜色外，还可以对它们的边框做色彩的设置，如果不需要边框时，还可以将边框设置为不可见（invisible）。例如：

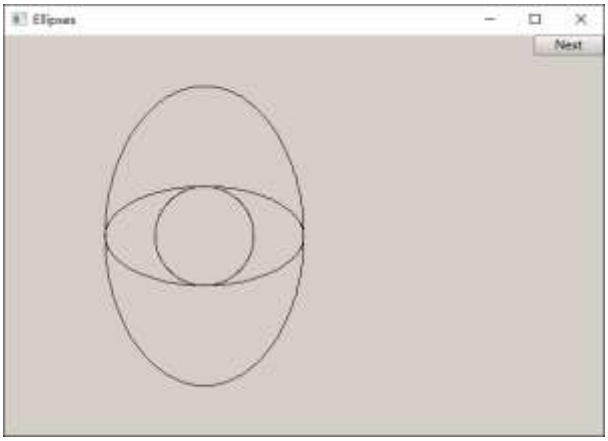
```
rect00.set_color(Color::invisible);
rect11.set_color(Color::invisible);
rect12.set_color(Color::invisible);
rect21.set_color(Color::invisible);
rect22.set_color(Color::invisible);
```

2.7 椭圆和圆

※椭圆由中心、长半轴和短半轴定义。

例如（chapter.13.13.cpp）：

```
    Ellipse e1 = {{200,200},50,50};
    Ellipse e2 = {{200,200},100,50};
Ellipse e3 = {{200,200},100,150};
```



椭圆利用极坐标方程来实现，的定义如下：

```
    struct Ellipse : Polar_function {
        Ellipse(Point p, int ww, int hh);

        Point center() const { return c; }
        int major() const { return w; }
        int minor() const { return h; }
        Point focus1() const;
```

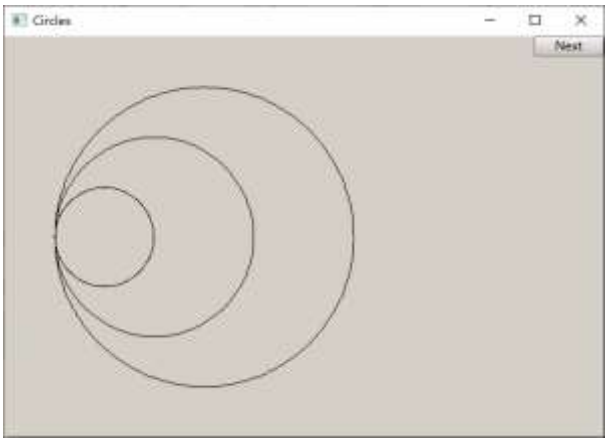
```
    Point focus2() const;

protected:
    Point c;
    int w, h;
};
```

※圆由圆心和半径来定义，是长轴和短轴相等的椭圆。

```
struct Circle : Ellipse {
    Circle(Point p, int rr) : Ellipse{p, rr, rr} { }
    int radius() const { return w; }
};
```

例（chapter.13.12.cpp）：

```
Circle c1 = {{100,200},50};
Circle c2 = {{150,200},100};
Circle c3 = {{200,200},150};
```



2.8 标记

※Marked_polyline用字符标记折线的顶点,例如（chapter.13.14.cpp）：

```
Marked_polyline mpl = {"1234",
    {{100,100}, {150,200}, {250,250}, {300,200}}};
```

Marked_polyline 的定义如下：

```
struct Marked_polyline : Open_polyline {
    Marked_polyline(const string& m, initializer_list<Point> lst);
    void draw_lines(Graphics& g) const;
private:
    string mark;
};
```

※Marks包含一组标记，此标记与线不相关联，例如，可以标记上例中的4个点而没有了线条显示（chapter.13.15.cpp）。

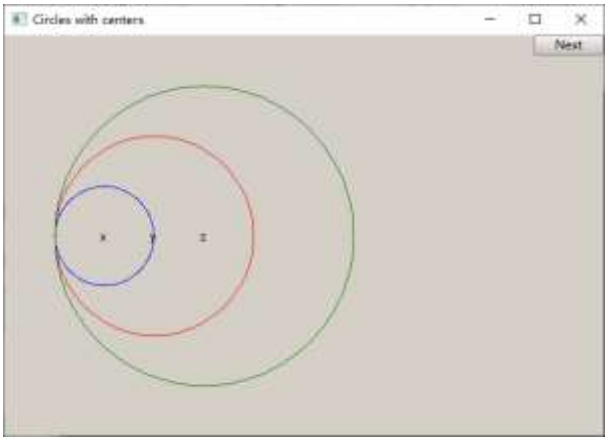
```
Marks pp {"x", {{100,100}, {150,200}, {250,250}, {300,200}}};
```

其实现只是简单地将 Maeked_polyline 的线段置为不可见。

```
struct Marks : Marked_polyline {
    Marked_polyline(const string& m, initializer_list<Point> lst)
        : Marked_polyline{m,lst}
    { set_color(Color::invisible); }
};
```

※Mark标记单个点，它由一个点和一个字符来表示

例如，对上面的圆标记圆心，可以如此操作（chapter.13.16 .cpp）：




```
Mark m1 = {{100,200},'x'};
```

```
Mark m2 = {{150,200},'y'};
Mark m3 = {{200,200},'z'};
```

其实，Mark 只不过是一个简单的只包含一个点的 Marks：

```
struct Mark : Marks {
    Mark(Point xy, char c) : Marks{string{1,c}} { add(xy); }
};
```

2.9 图片

※Image用来显示图片

※支持BMP和ICO两种图片格式。

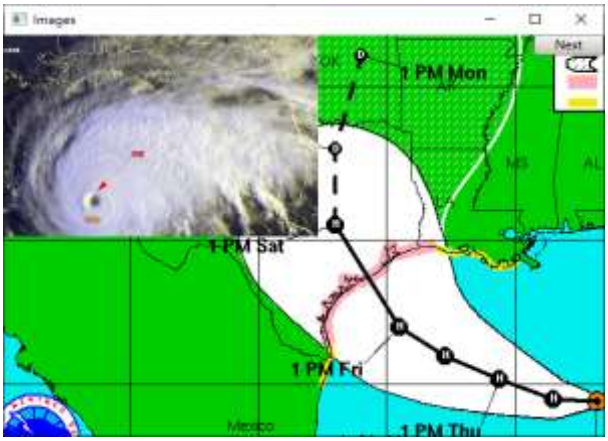
安装第三方库可支持JPG和PNG格式，具体方法参见Nana网站。

Image 的定义如下：

```
struct Image : Shape {
    Image(Point xy, const string& file);
    void open(const string& file);    // 打开图片文件
    // 打开内存中的图片数据
    void open(const void *data, std::size_t bytes);
    void draw_lines(Graphics& g) const;
    // 设置裁剪区域
    void set_mask(Point xy, int ww, int hh);
};
```

例（chapter.13.17.cpp）：

```
Image rita {{0,0},"rita.bmp"};
Image path {{0,0},"rita_path.bmp"};
path.set_mask({50,250},600,400);  // select likely landfall
win.attach(path);
win.attach(rita);
```



2.10 函数图

※Function用来绘制函数图，Function图形接口类的定义如下：

```
typedef double Fct(double);
```

```
struct Function : Shape { // 直角坐标函数
    // the function parameters are not stored
    Function(Fct f, double r1, double r2,
             Point orig, int count = 100,
             double xscale = 25, double yscale = 25);
};
```

Function 是一个 Shape，它的构造函数生成线段并把它们存储在 Shape 中。这些线段是对函数 f 的近似值，这个近似值是通过参数设置在范围 [r1,r2) 等间隔地计算了 count 次的 f 的值。参数 xscale 和 yscale 分别用来设定 X 坐标和 Y 坐标的比例。

※Polar_function用来绘制极坐标函数图，定义如下：

```
struct Polar_function : Shape {  // 极坐标方程
    // the function parameters are not stored
    Polar_function(Fct f, Point orig,          // 极坐标方程
             double t1 = 0, double t2 = 361, int count = 360,
             double xscale = 25, double yscale = 25);
};
```

参数 t1 和 t2 定义角度范围。

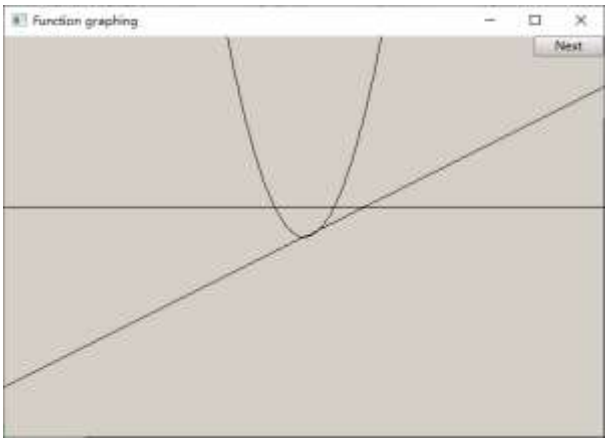
※Axis用来显示坐标轴，由一条线、在这条线上的一系列“刻度”和一个文本标签组成

```
struct Axis : Shape {
    enum Orientation { x, y, z };
    Axis(Orientation d, Point xy, int length,
         int number_of_notches=0, string label = "");

    void draw_lines(Graphics& g) const override;
    void move(int dx, int dy) override;
    void set_color(Color c);

    Text label;
    Lines notches;
};
```

※例：显示3个函数（chapter.15.2-4.cpp）

```
#include "Simple_window.h"
#include "Graph.h"

// 要显示的函数
double one(double) { return 1; }
double slope(double x) { return x/2; }
double square(double x) { return x*x; }

int main() {
    // 定义一组常数
```

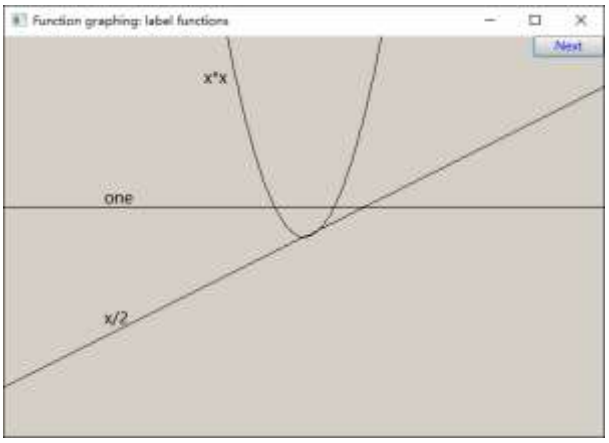
```
    const int xmax = 600;              // 窗口大小
    const int ymax = 400;

    const int x_orig = xmax/2;     // 原点定在窗口中心
    const int y_orig = ymax/2;
    const Point orig = {x_orig,y_orig};

    const double r_min = -10;      // 显示范围 [-10,10]
    const double r_max = 10;

    const int n_points = 400;      // 取样点数

    const double x_scale = 30;     // 比例因子
    const double y_scale = 30;

    Simple_window win0 = {{100,100},
           xmax, ymax, "Function graphing"};

    Function s1{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(s1);
    win0.attach(s2);
    win0.attach(s3);
    win0.wait_for_button();
}
```



※为3个函数添加标签

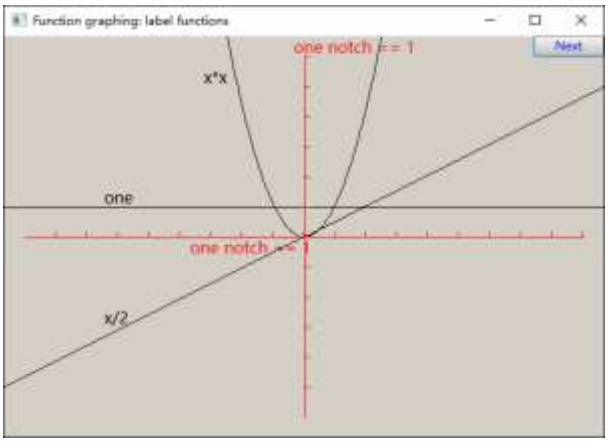
```
Text ts  = {{100,y_orig-40},"one"};
```

```
Text ts2 = {{100,y_orig+y_orig/2-20},"x/2"};
Text ts3 = {{x_orig-100,20},"x*x"};
win.set_label("Function graphing: label functions");
win.wait_for_button();
```



※添加坐标轴

```
// make the axis a bit smaller than the window
const int xlength = xmax-40;
const int ylength = ymax-40;

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

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



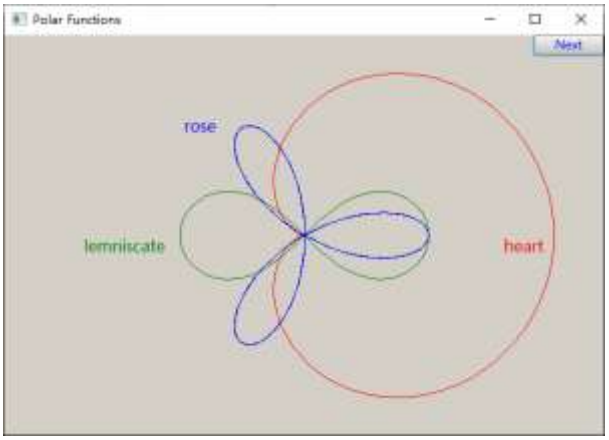
※Lambda表达式

仅仅为了在 Function 中使用一次就定义一个函数非常繁琐。C++提供了一种类似于匿名函数的表达式，可以作为参数，并能够执行函数的功能。例如我们可以这样定义 sloping_cos 的图形：

```
Function s5  = {[](double x) { return cos(x)+slope(x); },
               r_min,r_max,orig,400,30,30};
```

※例：显示极坐标函数（polar_func.cpp）

```
constexpr double a = 5, k = 3;
double heart(double t) { return a * (1+cos(t)); }
double lemniscate(double t) { return a * sqrt(fmax(cos(2*t), 0)); }
double rose(double t) { return a * cos(k*t); }

Polar_function s1 = {heart, orig};
Text ts1 = {orig + Point{200, 0}, "heart"};
s1.set_color(Colors::red);
ts1.set_color(Colors::red);

Polar_function s2 = {lemniscate, orig};
Text ts2 = {orig - Point{220, 0}, "lemniscate"};
s2.set_color(Colors::green);
ts2.set_color(Colors::green);

Polar_function s3 = {rose, orig};
Text ts3 = {orig - Point{120, 120}, "rose"};
s3.set_color(Colors::blue);
ts3.set_color(Colors::blue);
```



三 窗口和构件

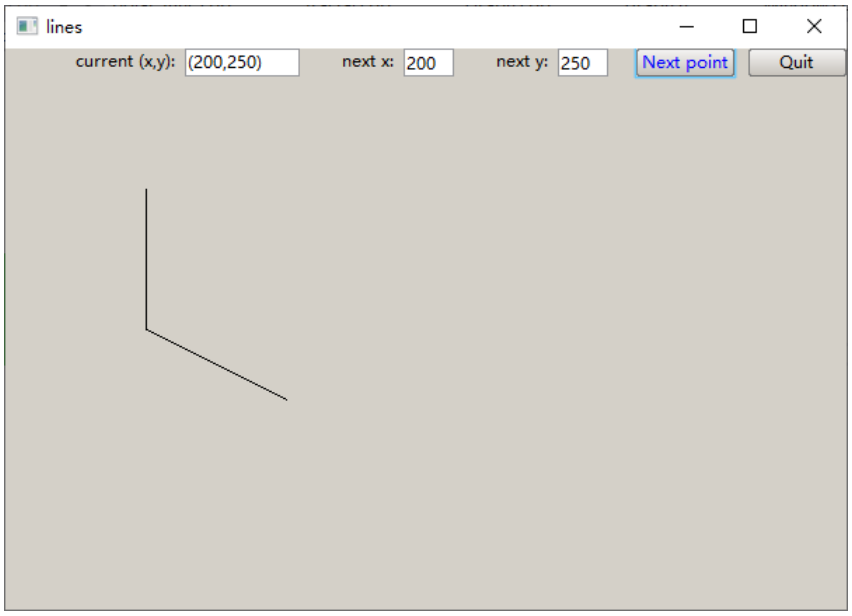
3.1 窗口

在图形界面中，所有的对象都必须在一个窗口上体现，窗口 Window 类的定义如下：

```
class Window : public nana::form {
public:
    // 由系统确定位置（放在屏幕中间）
    Window(int w, int h, const string& title);
    // xy 是窗口左上角位置
    Window(Point xy, int w, int h, const string& title);
```

```
    virtual ~Window() { }

    int x_max() const { return size().width;  }
    int y_max() const { return size().height; }
    void resize(unsigned w, unsigned h) { size({w, h}); }
    void set_label(const string& s) { caption(s); }

    void attach(Widget& w);
    void detach(Widget& w);
    void place(const char* fld, Widget& w);
// 从 nana::form 继承来的布局函数
//  void div(const char* div_text);
//  void collocate();

    void attach(Shape& s)     { dw.attach(s); }
    void detach(Shape& s)     { dw.detach(s); }
    void put_on_top(Shape& s) { dw.put_on_top(s); }
    void set_mouse_callback(Drawing::Mouse_cb cb);
    void set_size_callback (Drawing::Size_cb cb);
    void redraw();

private:
    Drawing dw;                 // Drawing object
    friend class Canvas;
    vector<Canvas*> canvases;  // Drawing widgets
};
```

通过 attach 和 detach 可以在窗口中添加或删除构件对象或图形对象。如下图所示，这是一个窗口的实例（chapter.16.5.cpp）。在这个窗口里包含了按钮、输入框、输出框和折线等对象，它们派生于构件类（Widget）以及前面介绍的图形类（Shape），下面对构件进行介绍。



3.2 构件基类

Widget是各种窗口构件的公共基类。

```
class Widget {
public:
    // 指定大小和位置
    Widget(Point xy, int w, int h, const string& s);
    Widget(const string& s); // 由布局确定大小和位置
    virtual ~Widget() { delete pw; }
    Widget& operator=(const Widget&) = delete; // 禁止拷贝
    Widget(const Widget&) = delete;

    const string& label() const { return lbl; }
    Point loc()   const { return pw->pos(); }
    int width()   const { return pw->size().width; }
    int height()  const { return pw->size().height; }

    void hide() { pw->hide(); }
    void show() { pw->show(); }
    void attach(Window& w)
    { if (pw) show(); else create_nana_widget(w); }
    virtual void set_fgcolor(Color cc) { pw->fgcolor(cc); }
    virtual void set_bgcolor(Color cc) { pw->bgcolor(cc); }
    virtual void set_font(Font fnt)  { pw->typeface(fnt); }

protected:
    nana::widget* pw = nullptr;          // 连接 Nana Widget
    // 在窗口中创建 Nana Widget 对象。每个具体的构件类需要定义自己的版本
    virtual nana::widget* create_nana_widget(nana::widget&) = 0;
private:
    string lbl;
};
```

为保持接口的简单性，Widget 并未从 nana::widget 继承，只是提供了访问 nana::widget 的接口。构件对象可以在创建时指定大小和位置，也可以由布局确定其大小和位置。构件对象通过 attach()添加到窗口对象中，可以通过 hide()和 show()来实现对象的隐藏和显示，并可设置构件的字体和颜色。

3.3 构件布局

在 3.1 节的例子中，构件的大小和位置在创建时指定。这种方法比较繁琐，且不能适应窗口大小的变化（chapter.16.5.cpp）。

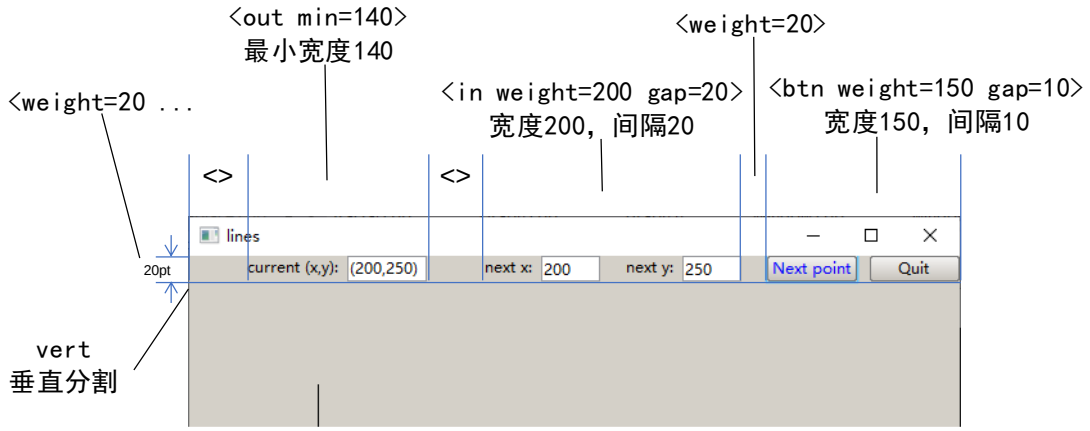
```
Lines_window::Lines_window(Point xy, int w, int h,
                           const string& title)
    : Window{xy,w,h,title},
    next_button{{x_max()-150,0},70,20,"Next point", /*...*/},
    quit_button{{x_max() - 70, 0}, 70, 20, "Quit", Exit},
```

```
    next_x{{x_max() - 360, 0}, 80, 20, "next x:"},
    next_y{{x_max() - 250, 0}, 80, 20, "next y:"},
    xy_out{{50, 0}, 160, 20, "current (x,y):"}
{ /*...*/ }
```

使用布局，可以更方便地实现同样的效果（chapter.16.5-place.cpp）。

```
Lines_window::Lines_window(Point xy, int w, int h,
                           const string& title)
    : Window{xy,w,h,title},
    next_button{"Next point", [this] { next(); }},
    quit_button{"Quit", Exit},
    next_x{"next x:"},
    next_y{"next y:"},
    xy_out{"current (x,y):"}
{
    div("vert<weight=20"
        "<><out min=140><>"
        "<in weight=200 gap=20><weight=20>"
        "<btn weight=150 gap=10>>"); // 划分布局域
    place("btn", next_button);       // 在域中放置构件
    place("btn", quit_button);
    place("in", next_x);
    place("in", next_y);
    place("out", xy_out);
    collocate();                     // 执行布局操作
    attach(lines);
}
```

首先调用 div 将窗口划分为布局域，再用 place 确定构件所属的域，最后调用 collocate 执行实际的布局操作。



div 使用一个字符串描述域的划分。域（field）由一对尖括号定义，可以任意嵌套。域可以有一个名字和一些属性。名字可以放在其他属性的前面，也可以放在后面。有一个隐含的顶层根域，它不必用尖括号定义。下表列举了部分常用的域属性。划分字符串的完整描述

可参见网页 https://github.com/cnjinhao/nana/wiki/Div-Text。

属性定义	说明
vert	该域包含的构件或子域垂直放置 若一个域无 vert 属性，则默认是水平放置
weight=50 weight=20%	域的宽度（水平放置）或高度（垂直放置）为 50 点 域的宽度或高度为上级域的 20%
min=100 max=200	域的最小宽度（或高度）为 100 域的最大宽度（或高度）为 200
arrange=[50,100]	该域包含的 2 个构件的宽度（或高度）分别为 50 和 100
gap=20	该域包含的构件之间有 20 点的间距
fit	根据域中包含的构件大小确定域的大小
grid=[3,2] collapse(0,1,2,1)	将域的空间划分为 3 列、2 行的网格 合并单元格：从第 0 列、第 1 行开始，2 列、1 行的 2 个单元格合并在一起 注：行列坐标从 0 开始计数

3.4 构件类

※标签

在窗口中显示文本。定义如下：

```
struct Label : Widget {
    Label(Point xy, int w, int h, const string& s,
        Format fm = Format::enable, Align al = Align::left);
    Label(const string& s,
        Format fm = Format::enable, Align al = Align::left);
    void set(const string&);   // 设置文本内容
private:
    nana::widget* create_nana_widget(nana::widget&) override;
};
```

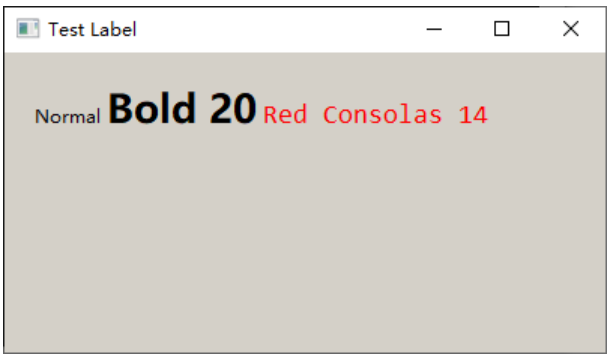
创建标签时可以指定是否支持格式定义以及对齐方式（左、中、右）。

可以用类似 HTML 的方式定义文本的字体、字号和颜色等属性。

详见 http://nanapro.org/en-us/documentation/widgets/label.htm

例（test_label.cpp）：

```
Label lbl = {{20, 20}, 360, 30,
    "Normal <bold size=20>Bold 20</> "
    "<red font=Consolas size=14>Red Consolas 14</>"};
```



※ 按钮

按钮在点击时会调用一个回调函数。定义如下：

```
struct Button : Widget {
    // 带参数的回调函数
    using callback = std::function<void(Button&)>;
    Button(Point xy, int w, int h, const string& s, callback cb)
        : Widget{xy, w, h, s}, cb_{cb}
    {}
    Button(Point xy, int w, int h, const string& s,
     std::function<void()> cb)        // 不带参数的回调函数
        : Button{ xy, w, h, s, [cb](Button&) { cb(); } }
    {}
    Button(const string& s, callback cb) : Widget{s}, cb_{cb} {}
    Button(const string& s, std::function<void()> cb)
        : Button{ s, [cb](Button&) { cb(); } }
    {}
private:
    nana::widget* create_nana_widget(nana::widget&) override;
    callback cb_;
};
```

按钮的回调函数可以不带参数，也可以带一个参数，用来接收触发事件的按钮对象。

※ 输入框和输出框

实现文本的输入和输出，定义如下：

```
struct In_box : Widget {
    In_box(Point xy, int w, int h, const string& s)
        : Widget(xy, w, h, s) { }
    In_box(const string& s) : Widget{s} { }
    int get_int();
    string get_string();
private:
    void create_nana_widget(Window& win) override;
};
struct Out_box : Widget {
    Out_box(Point xy, int w, int h, const string& s)
        : Widget(xy, w, h, s) { }
    Out_box(const string& s) : Widget{s} { }
    void put(int);
    void put(const string&);
private:
    nana::widget* create_nana_widget(nana::widget&) override;
};
```

In_box 用来接收用户输入的文本，在这里使用 get_string()将读入字符串文本或使用

get_int()读入整数；同时，可以使用 get_string()读取并检查是否得到了空字符串。

Out_box 用于向用户显示信息。与 In_box 相似，可以使用 put()实现输出字符串或整数。

※ 菜单

基于按钮的菜单，定义如下：

```
struct Menu : Widget {
    enum Kind { horizontal, vertical };
    Menu(Point xy, int w, int h, Kind kk, const string& label);
    Menu(Kind kk, const string& label);
    int attach(Button& b);     // attach Button to Menu
    int attach(Button* p);     // attach new Button to Menu

    Vector_ref<Button> selection;
    Kind k;
private:
    nana::widget* create_nana_widget(nana::widget&) override;
};
```

菜单本质上是一组按钮。Point xy 表示菜单左上角的位置，(w, h)表示一组按钮的总大小。每个菜单按钮是一个独立的构件，作为 attach()的参数提供给菜单。

※画布

Canvas 在窗口中提供一个画图的区域，可以附加图形对象。定义如下：

```
class Canvas : public Widget {
public:
    Canvas(Point xy, int w, int h) : Widget{xy, w, h, ""} { }
    Canvas() : Widget{""} { }

    using Widget::attach;
    void attach(Shape& s);
    void detach(Shape& s);
    void put_on_top(Shape& p);
    void redraw();
    void set_mouse_callback(Drawing::Mouse_cb cb);
    void set_size_callback(Drawing::Size_cb cb);
private:
    nana::widget* create_nana_widget(nana::widget&) override;
};
```

若将图形对象附加于窗口，其坐标位置是相对于窗口的；而将图形对象附加于画布对象时，其坐标位置是相对于画布对象的。这样可以利用构件布局来定位图形对象。

※动画

Animation 以一定的帧速率播放一系列的帧图形。定义如下：

```
class Animation : public Widget {
public:
```

```
    Animation(Point xy, int w, int h,
              bool looped = false, int fps = 23);
    Animation(bool looped = false, int fps = 23);

    bool is_looped() const { return looped_; }
    int  fps() const { return fps_; }
    // put frame pictures at the end
    void push_back(const std::vector<string>& img_files);
    // put frame shapes at the end
    void push_back(const std::vector<const Shape*>& shapes);
    void play();
    void pause();
private:
    nana::widget* create_nana_widget(nana::widget&) override;
    bool looped_;
    int  fps_;
};
```

在构造函数中可指定帧速率以及是否循环播放。要播放的帧图形由 push_back()加入，可以是一组图片文件，也可以是一组已定义的图形对象。目前版本的 Animation 只能调用一次 push_back()，若调用了多次，仅第一次加入的帧图形有效。play()开始播放，pause()暂停播放。

完整的应用实例详见 test_animation.cpp。

3.5 响应事件

按钮可以响应点击事件，点击按钮时调用其回调函数。此外还可以为窗口或画布对象注册回调函数，以响应鼠标事件、键盘事件和尺寸变化事件。

鼠标事件定义如下，包括事件类型、位置、按键、滚轮和功能键等信息。

```
struct Mouse_event {
    enum Event_type { dbl_click, push, drag, release, move, wheel };
    enum Buttons { any, left, middle, right };
    enum Wheels { vertical, horizontal };
    enum States {              // masks for state
        s_left=0x01, s_middle=0x02, s_right=0x04, s_buttons=0x07,
        s_alt =0x10, s_shift =0x20, s_ctrl=0x40
    };
    Event_type type;
    Point pos;
    Buttons button;
    int state;
    Wheels which = vertical;
    int distance = 0;
};
using Mouse_cb = std::function<void(const Mouse_event& e)>;
```

键盘事件定义如下，包含事件类型、按键和修饰键（alt, ctrl, shift）等信息。

```
struct Keyboard_event {
    event_code evt_code;    // key_press or key_release
    wchar_t key;            // the key
    bool alt, ctrl, shift;  // the modifier keys
    ... ...
};
using Keyboard_cb = std::function<void(const Keyboard_event& e)>;
```

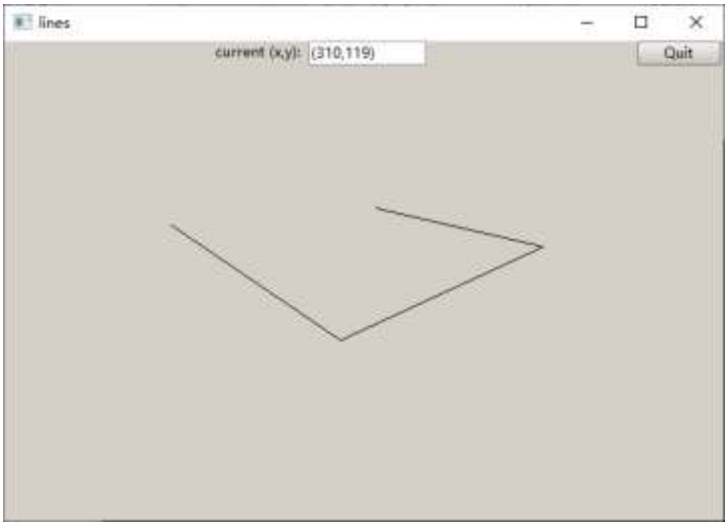
尺寸变化事件定义如下，包含变化后的宽度与高度等信息。

```
struct Size_event {
    unsigned width, height;
    ... ...
};
using Size_cb = std::function<void(const Size_event& e)>;
```

鼠标回调函数接收一个鼠标事件作为参数；键盘回调函数接收一个键盘事件作为参数；尺寸回调函数接收一个尺寸变化事件作为参数。若回调函数注册在窗口上，它接收到的鼠标位置或尺寸信息是相对于窗口的；若回调函数注册在画布对象上，它接收到的鼠标位置或尺寸信息是相对于画布对象的。键盘事件的应用实例详见 test_keyboard.cpp，鼠标事件与尺寸变化事件的应用实例详见 chapter.16.5-mouse.cpp 和 fractal.cpp。

3.6 GUI 编程实例

这是 3.1 节例子的改进版本（chapter.16.5-mouse.cpp）。用户通过点击鼠标在窗口中绘制一条折线。current(x,y)输出框显示用户在绘图区域（画布）中点击的位置。点击“Quit”按钮可以退出程序。



※窗口类型定义：

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
private:
    Open_polyline lines;
```

```
    Button quit_button;
    Out_box xy_out;
    Canvas canvas;
    void mouse_cb(const Mouse_event& evt);
    void out_pos(Point pt);
};
```

窗口 Lines_window 派生于 Window 类，包含一个 Open_polyline 对象，用于绘制折线；一个退出按钮（quit）；一个输出框，显示当前光标的坐标值；一个画布对象，用来放置图形。

Lines_window 的构造函数用于初始化所有构件对象和图形对象，并对构件进行布局。

```
Lines_window::Lines_window(Point xy, int w, int h,
                           const string& title)
    : Window{xy, w, h, title},
      quit_button{"Quit", Exit},
      xy_out{"current (x,y):"}
{
    div("vert<weight=20<><out min=135><>"
        "<quit weight=70>><draw>");
    place("quit", quit_button);
    place("out", xy_out);
    place("draw", canvas);
    collocate();
    canvas.attach(lines);
    canvas.set_mouse_callback(
        [this](const Mouse_event& evt) { mouse_cb(evt); });
}
```

画布对象使用 Lambda 表达式注册了一个鼠标回调函数，功能是调用 Lines_window 的 mouse_cb 成员函数实现绘图操作。“Quit”按钮的回调函数是预先定义的 Exit()函数，功能是关闭所有窗口。

※mouse_cb()函数：

```
void Lines_window::mouse_cb(const Mouse_event& evt) {
    if (evt.type == Mouse_event::push)
        lines.add(evt.pos);
    else if (evt.type == Mouse_event::drag) {
        auto last = lines.number_of_points() - 1;
        if (last < 0) return;
        if (last == 0) lines.add(evt.pos);
        else lines.set_point(last, evt.pos);
    }
    else return;
    out_pos(evt.pos);
    redraw();
```

```
}
```

该函数在鼠标点击时为折线对象添加一个线段。在按住鼠标拖动时，实现橡皮筋拖动的效果。将新加入的端点坐标输出到输出框中，并调用 redraw()函数更新窗口内容。

※out_pos ()函数

输出当前坐标。

```
void Lines_window::out_pos(Point pos) {
    xy_out.put("(" + to_string(pos.x) + ","
                + to_string(pos.y) + ")");
}
```

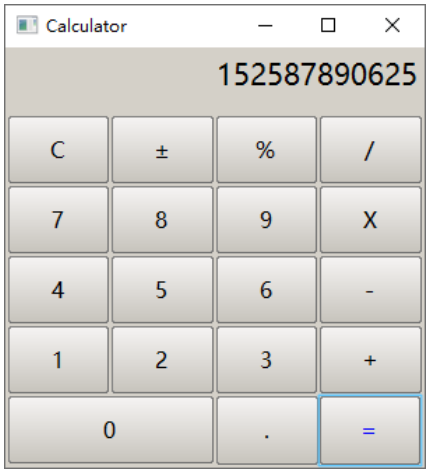
※main()函数

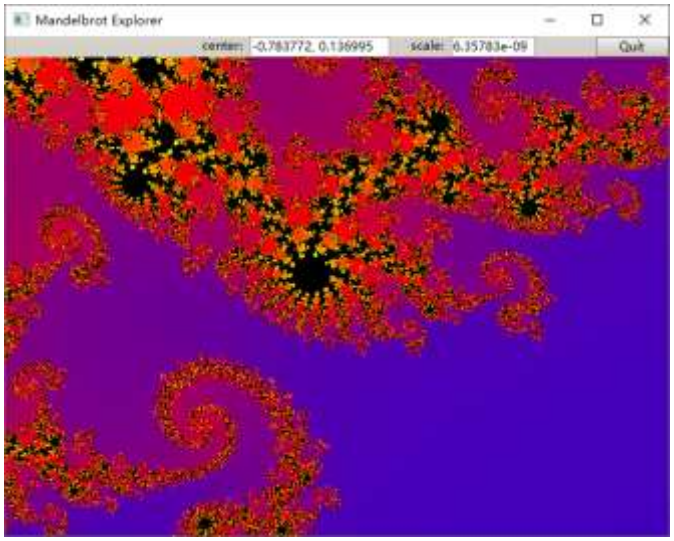
```
#include "GUI.h"
int main(){
    Lines_window win {{100,100}, 600, 400, "lines"};
    gui_main(win);
}
```

main()函数非常简单，只需实例化一个 Lines_window 窗口，调用 gui_main()显示窗口，并将程序运行的控制权交给 GUI 系统。

※其他复杂例子

1. 计算器（calculator.cpp）：网格式构件布局。
2. 分型图形（fractal.cpp）：绘制分型图形，利用鼠标事件缩放图形。





附：本课程图形库与《C++程序设计原理与实践》书中原始版本的不同

本课程图形库基于 Nana，与 PPPC 基于 FLTK 的原始版本相比，绘图功能有所减弱，而 GUI 功能明显增强。

1. Nana 是一个简单的 GUI 库，绘图功能比较弱，不能支持原始版本的部分绘图功能：
   不支持线形；填充色只对矩形有效，对于多边形等复杂形状无效。

2. 颜色名称由嵌套在 Color 类中的枚举类型改为枚举类类型 Colors。可以使用 RGB 值定义颜色。例：
   原版本：`poly.set_color(Color::red);`
   新版本：`poly.set_color(Colors::red);`
   `poly.set_color({255, 0, 0});`

3. 字体名称使用字符串类型代替枚举类型，可以定义安装过的任意字体。字号并入 Font 类。例：
   原版本：`t.set_font(Font::times_bold); t.set_font_size(20);`
   新版本：`t.set_font({"Times", true, false, 20});`

4. 支持的图片格式为 BMP 和 ICO；原始版本则是 JPG 和 GIF。
   安装第三方库可支持 JPG 和 PNG 格式，具体方法参见 Nana 网站。

5. 新增极坐标函数图，并以此为基础定义椭圆和圆。

6. gui_main 的原型由 `int gui_main();`改为 `void gui_main(Window&);`接受的参数是要显示的窗口。

7. 简化了回调函数的定义。例：
   原版本：

```
startButton{{100, 300}, 100, 40, "Start",
        [](Address, Address win)
        { reference_to<HangmanWindow>(win).newGame(); }},
quitButton{{300, 300}, 100, 40, "Quit",
        [](Address, Address win)
        { reference_to<HangmanWindow>(win).hide(); }},
… …
letterButton.push_back(
    new Button{Point{170+j*45, 10+i*55}, 35, 45, string(1, 'a'+c),
    [](Address wigdet, Address win)
    { reference_to<HangmanWindow>(win).
      selectLetter(reference_to<Fl_Widget>(wigdet).label()); }
});
```

新版本：

```
startButton{{100, 300}, 100, 40, "Start", [this] { newGame(); }},
quitButton{{300, 300}, 100, 40, "Quit", Exit },
… …
letterButton.push_back(
    new Button{Point{170+j*45, 10+i*55}, 35, 45, string(1, 'a'+c),
        [this] (Button& b) { selectLetter(b.label); }
});
```

8. 新增构件布局功能。新增 Label 和 Canvas 构件，以充分利用布局功能。

9. 新增动画构件（Animation）。

10. 可以为窗口和画布对象注册鼠标回调函数和尺寸回调函数，并为窗口注册键盘回调函数。

11. 原始版本中 In_box 和 Out_box 在构造函数中指定的大小只是输入框的大小，不包括左侧的文本标签；在新版本中则是包括标签在内的大小。
    原始版本中 Menu 在构造函数中指定的大小只是一个按钮的大小；在新版本中则是整个按钮组大小。