附录 E　GUI 实现

“当你最终理解了你在做什么时，事情就会向正确的方向发展了。”

——Bill Fairbank

本附录介绍回调函数、Window、Widget 和 Vector_ref 的实现细节。在第 16 章中，由于还没有介绍指针和类型转换的相关知识，我们无法给出 GUI 实现的完整描述，因此将这部分内容放到本附录中。

E.1　回调函数实现

回调函数可实现如下代码：

```
void Simple_window::cb_next(Address, Address addr)
        // call Simple_window::next() for the window located at pw
{
        reference_to<Simple_window>(addr).next();
}
```

如果你已经理解了第 17 章的内容，很显然 Address 必须是 void* 。而且，当然，reference_to < Simple_window > (addr) 必须是由名为 addr 的 void* 创建的 Simple_window 的引用。但是，除非你以前有相关的程序设计经验，否则在阅读第 17 章之前，你不会有“很显然”或者“当然”的感觉。因此，让我们好好看一下地址使用的细节。

如附录 A. 17 所述，C++ 提供了类型命名的功能。例如：

```
typedef void* Address;      // Address is a synonym for void*
```

这意味着现在就可以用名字 Address 来代替 void* 了。在此，我们用 Address 这个名字来强调传递了一个地址，并掩盖这样一个事实：void* 是指向未知类型的对象的指针。

因此，cb_next( ) 接受一个名为 addr 的 void* 参数，并立即用某种方法将其转换为 Simple_window&：

```
reference_to<Simple_window>(addr)
```

reference_to 是一个模板函数(参见附录 A. 13)：

```
template<class W> W& reference_to(Address pw)
        // treat an address as a reference to a W
{
        return *static_cast<W*>(pw);
}
```

此处，我们将模板函数设计为类似一个类型转换——它将 void* 转换为 Simple_winodow&。类型转换操作 static_cast 在 17. 8 节中有详细描述。

编译器无法验证 addr 是否指向一个 Simple_window 对象，但语言规则要求编译器此时信任程序员。幸运的是，我们是正确的。之所以确定我们是正确的，是因为 FLTK 会将我们传递给它的指针传递回来。由于我们知道传递给 FLTK 的指针是什么类型，因此可以使用 reference_to 将其“取回来”。这种方法有些混乱，也未经检查，但在系统底层并不罕见。

一旦你拥有了一个 Simple_window 的引用，就可以使用它来调用 Simple_window 的成员函数。例如(参见 16. 3 节)：

```
void Simple_window::cb_next(Address, Address pw)
        // call Simple_window::next() for the window located at pw
{
        reference_to<Simple_window>(pw).next();
}
```

我们简单地使用了一个混乱的回调函数 cb_next( ) 来调整类型，以便调用一个普通的成员函数 next( )。

E.2　Widget 实现

Widget 接口类如下所示：

```
class Widget {
        // Widget is a handle to a Fl_widget — it is *not* a Fl_widget
        // we try to keep our interface classes at arm's length from FLTK
public:
        Widget(Point xy, int w, int h, const string& s, Callback cb)
                :loc(xy), width(w), height(h), label(s), do_it(cb)
{ }

        virtual ~Widget() { }       // destructor

        virtual void move(int dx,int dy)
                { hide(); pw->position(loc.x+=dx, loc.y+=dy); show(); }
        virtual void hide() { pw->hide(); }
        virtual void show() { pw->show(); }

        virtual void attach(Window&) = 0;   // each Widget defines at least
                                            // one action for a window

        Point loc;
        int width;
        int height;
        string label;
        Callback do_it;

protected:
        Window* own;      // every Widget belongs to a Window
        Fl_Widget* pw;    // a Widget "knows" its Fl_Widget
};
```

注意，Widget 会跟踪 FLTK widget 及关联的 Window 对象。注意，这需要使用指针，因为一个 Widget 在其生命期内可能与不同的 Windows 对象相关联。引用或者命名对象是不能达到要求的(考虑一下为什么)。

每个 Widget 有一个位置(loc)、一个矩形形状(width 和 height)和一个标签(label)。有趣的是，它还有一个回调函数(do_it)，这个回调函数将屏幕上 Widget 的图像与代码连接起来。其他操作(如 move( )、show( )、hide( )和 attach( ))的含义是显然的。

Widget 看起来还只是“半成品”。其设计目标是：如果就是作为一个实现类，用户不必经常查看其细节。它很适合于重新设计。我们对所有公有数据成员存有疑问，而那些“显然”的操作通常需要针对意料之外的细微问题进行重新检查。

Widget 包含虚函数，可以作为基类，因此它有一个虚析构函数(参见 17.5.2 节)。

E.3　Window 实现

我们应该在什么时候使用指针，又应该在什么时候使用引用呢？在 8.5.6 节中，我们对这方

面的一些一般性问题进行了讨论。在本附录中，我们只是注意到，一些程序员喜欢使用指针，而当需要在程序中在不同时刻引用不同对象时，应该使用指针。

到目前为止，我们还未展示图形和 GUI 库中的一个中心类——Window。最主要的原因是它使用了指针，而且其实现使用了基于自由内存空间分配的 FLTK。下面是 Window. h 中的 Window 类定义：

```
class Window : public Fl_Window {
public:
    // let the system pick the location:
    Window(int w, int h, const string& title);
    // top left corner in xy:
    Window(Point xy, int w, int h, const string& title);

    virtual ~Window() { }

    int x_max() const { return w; }
    int y_max() const { return h; }

    void resize(int ww, int hh) { w=ww, h=hh; size(ww,hh); }

    void set_label(const string& s) { label(s.c_str()); }

    void attach(Shape& s) { shapes.push_back(&s); }
    void attach(Widget&);

    void detach(Shape& s);          // remove w from shapes
    void detach(Widget& w);         // remove w from window
                                    // (deactivates callbacks)

    void put_on_top(Shape& p);  // put p on top of other shapes
protected:
    void draw();
private:
    vector<Shape*> shapes;    // shapes attached to window
    int w,h;                  // window size

    void init();
};
```

这样，当 attach( )一个形状时，我们在 shapes 中保存一个指针，Window 对象即可利用它来绘出图形。由于允许随后 detach( )这个形状，因此需要一个指针。基本上，被 attach( )的形状还是由用户代码拥有，我们只是传递给 Window 对象一个引用而已。Window :: attach( )将其参数转换为一个指针，方便存储。如上所示，attach( )很简单，detach( )稍微复杂些。查看 Windows. cpp，我们发现：

```
void Window::detach(Shape& s)
    // guess that the last attached will be first released
{
    for (unsigned int i = shapes.size(); 0<i; --i)
        if (shapes[i-1]==&s) shapes.erase(&shapes[i-1]);
}
```

成员函数 erase( )从 vector 中删除(“擦除”)一个值，将 vector 的规模减小 1(参见 20. 7. 1 节)。

Window 的设计意图就是要作为一个基类来使用，因此它有一个虚析构函数(参见 17. 5. 2 节)。

E. 4　Vector_ref

基本上，Vector_ref 是模拟引用的 vector。你可以用引用或指针来初始化一个 Vector_ref：

- 如果将一个对象以引用方式传递给 Vector_ref，则假定调用者会负责对象的生命期（例如，对象是作用域之内的变量）。
- 如果对象以指针方式传递给 Vector_ref，则假定它是用 new 分配的，由 Vector_ref 负责释放空间。

元素以指针而不是对象副本的形式存入 Vector_ref，其访问遵循引用语义。例如，你可以将一个 Circle 放入 Vector_ref < Shape > 中，而不会面临子数组问题：

```
template<class T> class Vector_ref {
    vector<T*> v;
    vector<T*> owned;
public:
    Vector_ref() {}
    Vector_ref(T* a, T* b = 0, T* c = 0, T* d = 0);

    ~Vector_ref() { for (int i=0; i<owned.size(); ++i) delete owned[i]; }

    void push_back(T& s) { v.push_back(&s); }
    void push_back(T* p) { v.push_back(p); owned.push_back(p); }

    T& operator[](int i) { return *v[i]; }
    const T& operator[](int i) const { return *v[i]; }

    int size() const { return v.size(); }
};
```

Vector_ref 的析构函数释放每个以指针传递来的对象。

E.5　实例：Widget 操作

下面是一个完整的示例程序，它演示了很多 Widget/Window 特性。代码中只给出了很少的注释。不幸的是，在现实中像这样缺乏必要注释的情况并不罕见。作为一个练习，请将这个程序运行起来，并给出必要的注释。

当你运行这个程序时，它会定义 4 个按钮：

```
#include "../GUI.h"
using namespace Graph_lib;

class W7 : public Window {
    // four ways to make it appear that a button moves around:
    // show/hide, change location, create new one, and attach/detach
public:
    W7(int w, int h, const string& t);

    Button* p1;         // show/hide
    Button* p2;
    bool sh_left;

    Button* mvp;        // move
    bool mv_left;

    Button* cdp;        // create/destroy
    bool cd_left;
    Button* adp1;       // activate/deactivate
    Button* adp2;
    bool ad_left;
```

```
        void sh();              // actions
        void mv();
        void cd();
        void ad();

        static void cb_sh(Address, Address addr)     // callbacks
                { reference_to<W7>(addr).sh(); }
        static void cb_mv(Address, Address addr)
                { reference_to<W7>(addr).mv(); }
        static void cb_cd(Address, Address addr)
                { reference_to<W7>(addr).cd(); }
        static void cb_ad(Address, Address addr)
                { reference_to<W7>(addr).ad(); }
};
```

但是，W7(7 号 Window 实验)实际包含 6 个按钮，它将其中两个隐藏了起来：

```
W7::W7(int w,int h, const string& t)
        :Window(w,h,t),
        sh_left(true), mv_left(true), cd_left(true), ad_left(true)
{
        p1 = new Button(Point(100,100),50,20,"show",cb_sh);
        p2 = new Button(Point(200,100),50,20,"hide",cb_sh);

        mvp = new Button(Point(100,200),50,20,"move",cb_mv);

        cdp = new Button(Point(100,300),50,20,"create",cb_cd);

        adp1 = new Button(Point(100,400),50,20,"activate",cb_ad);
        adp2 = new Button(Point(200,400),80,20,"deactivate",cb_ad);

        attach(*p1);
        attach(*p2);
        attach(*mvp);
        attach(*cdp);
        p2->hide();
        attach(*adp1);
}
```

程序中使用了 4 个回调函数。每个回调函数使你按下的按钮消失，并显示一个新的按钮。但是，这一效果是经过 4 个步骤来实现的：

```
void W7::sh()       // hide a button, show another
{
        if (sh_left) {
                p1->hide();
                p2->show();
        }
        else {
                p1->show();
                p2->hide();
        }
        sh_left = !sh_left;
}

void W7::mv()       // move the button
{
        if (mv_left) {
                mvp->move(100,0);
        }
        else {
                mvp->move(-100,0);
```

```
        }
        mv_left = !mv_left;
}

void W7::cd()    // delete the button and create a new one
{
        cdp->hide();
        delete cdp;
        string lab = "create";
        int x = 100;
        if (cd_left) {
                lab = "delete";
                x = 200;
        }
        cdp = new Button(Point(x,300), 50, 20, lab, cb_cd);
        attach(*cdp);
        cd_left = !cd_left;
}

void W7::ad()    // detach the button from the window and attach its replacement
{
        if (ad_left) {
                detach(*adp1);
                attach(*adp2);
        }
        else {
                detach(*adp2);
                attach(*adp1);
        }
        ad_left = !ad_left;
}

int main()
{
        W7 w(400,500,"move");
        return gui_main();
}
```

这个程序演示了在窗口中添加和去掉 Widget 或者是仅仅显示 Widget 的基本方法。