Programming With Nana C++ Library

Jinhao

Draft Updated: February 20, 2012

Comments and feedback appreciated!

http://stdex.sourceforge.net
http://nanaproject.wordpress.com
cnjinhao@hotmail.com

Content

1, Getting Started

This chapter shows how to create a small graphical user interface(GUI) application with Nana C++ Library.

1.1, Hello Nana

Let’s start with a simple program, we will study it line by line.

```
1    #include <nana/gui/wvl.hpp>
2    #include <nana/gui/widgets/label.hpp>

3    int main()
4    {
5        nana::gui::form form;
6        nana::gui::label label(form, 0, 0, 100, 20);
7        label.caption(STR("Hello Nana"));
8        form.show();
9        nana::gui::exec();
     }
```

Lines 1 and 2 include the definitions of class form and class label in namespace nana::gui. Additionally, every GUI application written by Nana C++ Library must include the header file nana/gui/wvl.hpp.

Line 5 defines a nana::gui::form object, it is a window used for placing a label widget in this example.

Line 6 defines a nana::gui::label object to display a text string. The label object is created in the form. A widget is a visual element in a user interface.

Line 7 sets the caption of label object. Every widget has a caption for displaying a title. In this line, there is a string within a macro named STR, this macro is a string wrapper used for easy switch the application between UNICODE and ASCII version.

Line 8 makes the form visible.

Line 9 passes the control of the application on to Nana. At this point, the program enters the event loop for waiting for and receiving a user action, such as mouse move, mouse click and keyboard press. The function nana::gui::exec blocks till form is destroyed, and the example demonstrate is that program exits when a user closes the window.

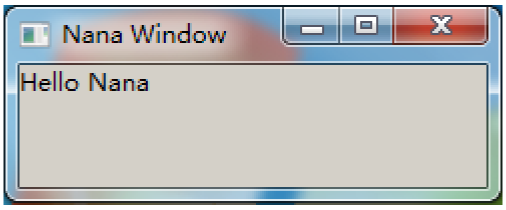


Figure 1.1 Hello Nana

Now, you can run the program on your own machine. But firstly you should have Nana C++ Library installed in your system. A method to install is explained in Installation Library Documentation in Nana Programmer's Guide.

1.2, Making An Event

The second example shows how to respond a user action. To receive a user action, an event handler should be registered to a widget. Nana waits for a user action and invokes the event handler of corresponding event. The application consists of a button that the user can click to quit.

```
#include <nana/gui/wvl.hpp>
#include <nana/gui/widgets/button.hpp>

int main()
{
    using namespace nana::gui;
    form fm;
    button btn(fm, 0, 0, 100, 20);
    btn.caption(STR("Quit"));
    btn.make_event<events::click>(API::exit);
    fm.show();
    exec();
}
```

This source code is similar to Hello Nana, except that we are using a button instead of a label, and we are making an event to respond a user click. Every widget class has a set of methods to make events, they are named "make_event", this is a function template, the first template parameter specifying what event must be given explicitly when calling the function, and the parameter of the member function is an event handler. As above code shown, we make the API::exit() as an event handler for the button's click event. The exit() function is an API in Nana C++ Library, it closes all windows in current GUI thread and terminates the event loop. it will be called when the user clicks the button.

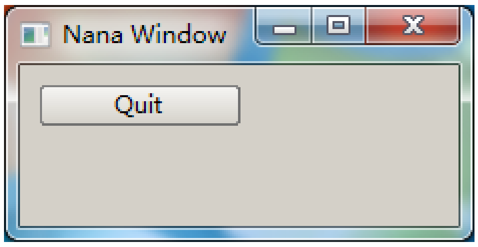


Figure 1.2 Event Example

2, Function Objects

This chapter shows two basic concepts: function object and lambda, and function object is a requirement to understand Nana C++ Library.

A Function Object, or Functor (the two terms are synonymous) is simply any object that can be called as if it is a function, more generally, so is an object of a class that defines function call operator().

The function object is an impressive technology. A function object is a more general concept than a function because a function object can have state that persist across several calls and can be initialized and examined from outside the object, unlike a static local variable. For example:

```
class sum
{
public:
    sum() : i_(0)
    {}

    operator int() const volatile
    {
        return i_;
    }

    //this makes the objects of this class can be used like a function.
    void operator()(int x) volatile
    {
        i_ += x;
    }
private:
    int i_;
};
```

```
void foo(const std::vector<int>& v)
{
    //gather the sum of all elements.
    std::cout<<std::for_each(v.begin(), v.end(), sum())<<std::endl;
}
```

std::for_each() returns a copy of the object of sum, and we are able to retrieve the state or result. On the basis of this feature that function objects retain its own state, it is easy used for concurrency process, and it is extensively used for providing flexibility in the library implementation.

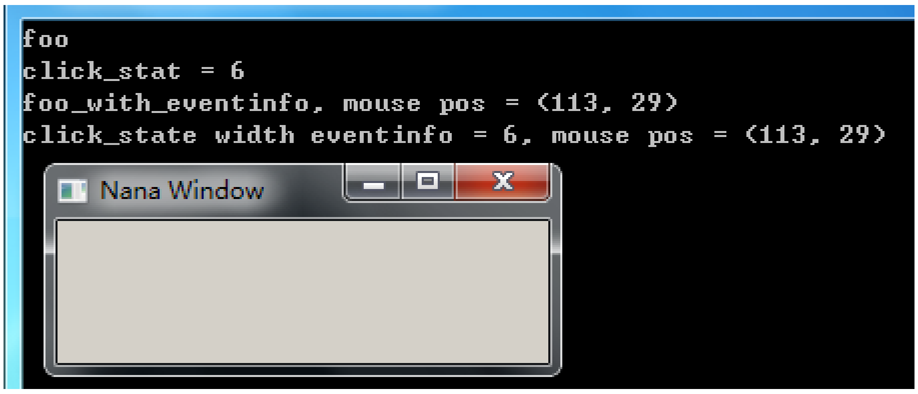
Nana C++ Library uses a large number of function objects to make the framework work. To make the framework flexible enough, Nana C++ Library introduced a general functor class template:

```
template<typename Ftype> class nana::functor;
```

The template parameter Ftype is specified what function type the functor delegates. By using the functor class template, we are able to make Nana C++ Library get rid of types that involved. For example:

```
#include <nana/gui/wvl.hpp>
#include <iostream>

void foo()
{
    std::system("cls");
    std::cout<<"foo"<<std::endl;
}

void foo_with_eventinfo(const nana::gui::eventinfo& ei)
{
    std::cout<<"foo_with_eventinfo, mouse pos = ("
        <<ei.mouse.x<<", "<<ei.mouse.y<<")"
        <<std::endl;
}

class click_stat
{
public:
    click_stat(): n_(0)
    {}

    void respond()
    {
        std::cout<<"click_stat = "<<++n_<<std::endl;
    }

    void respond_ei(const nana::gui::eventinfo& ei)
    {
        std::cout<<"click_state width eventinfo = "<<n_
            <<", mouse pos = ("
            <<ei.mouse.x<<", "<<ei.mouse.y<<")"
            <<std::endl;
    }
private:
    int n_;
};


int main()
{
    using namespace nana::gui;
    typedef nana::functor<void()> fun_t;
    typedef nana::functor<void(const eventinfo&)> fun_with_param_t;

    form fm;
    click_stat cs;

    fun_t f(foo);
    fm.make_event<events::click>(f);

    f = fun_t(cs, &click_stat::respond);
    fm.make_event<events::click>(f);

    fun_with_param_t fp(foo_with_eventinfo);
    fm.make_event<events::click>(fp);

    fp = fun_with_param_t(cs, &click_stat::respond_ei);
    fm.make_event<events::click>(fp);

    fm.show();
    exec();
}
```

There are four different types of event handlers can be processed by using nana::functor class template. It is flexible and reducing the complexity of study and use.



```
foo
click_stat = 6
foo_with_eventinfo, mouse pos = (113, 29)
click_state width eventinfo = 6, mouse pos = (113, 29)
```

Figure 2.1 Various methods to make events to respond click.

In the previous example, we illustrated the use of nana::functor and the flexibility of using a function object. In practically, creating a nana::functor object is not required. Using these functions in this way instead of creating a nana::functor, like this.

```
int main()
{
    using namespace nana::gui;

    form fm;
    click_stat cs;

    fm.make_event<events::click>(foo);
    fm.make_event<events::click>(cs, &click_stat::respond);

    fm.make_event<events::click>(foo_with_eventinfo);
    fm.make_event<events::click>(cs, &click_stat::respond_ei);

    fm.show();
    exec();
}
```

2.1, Predefined Function Objects

Nana C++ Library includes many different predefined function objects. Using these function objects together with function templates increases the expressive power of the library as well as making the resulting code much more efficient. For example, if a C++ program wants to close the form when the form is being clicked.

form.make_event<events::click>(destroy(form));
Please include <nana/gui/functional.hpp> before using these function objects.

```
class destroy
{
public:
    destroy(nana::gui::window wd);
    void operator()() const;
};
```

Destroy the window.

```
class hide
{
public:
    hide(nana::gui::window wd);
    void operator()() const;
};
```

Hide the window.

```
class show
{
public:
    show(nana::gui::window wd);
    void operator()() const;
};
```

Show the window.

2.2, The Lambda Expression

A lambda expression is a mechanism for specifying a function object. The lambda is a new feature that introduced into C++ language recently, the primary use for a lambda is to specify a simple action to be performed by some functions. For example:

```
#include <nana/gui/wvl.hpp>
#include <iostream>

int main()
{
    nana::gui::form form;
    form.make_event<nana::gui::events::click>(
        []{ std::cout<<"form is clicked"<<std::endl; }
        );
    form.show();
    nana::gui::exec();
}
```

The argument `[]{ std::cout<<"form is clicked"<<std::endl; }` is a "lambda" (or "lambda function" or "lambda expression") in C++ language(C++11). A lambda starts with plain [], and compound-state block {} defines a function body. In fact, A lambda defines an anonymous function object, and therefore a lambda could be invoked through a function-call syntax.

```
[]{ std::cout<<"hello, Nana"<<std::endl; }();
```

The use of lambda is creating an anonymous function object and so the arguments should be specified. For example:

```
form.make_event<nana::gui::events::click>(
        [](const nana::gui::eventinfo& ei)
        {
            std::cout<<"mouse pos=("
                      <<ei.mouse.x<<", "<<ei.mouse.y
                      <<std::endl;
        }
    );
```

The lambda-declarator () is used like a parameter-list. Let's stop the introduction to the lambda, if you want more details of lambda, please refer to other C++ books.

2.3, Think about the Design

Assuming the design of a framework that generates some data and it requires a module for representing these data in GUI. In general, we just introduce an interface for outputting these data, for example:

```
struct data
{
    std::string url;
};

class uiface
{
public:
    virtual ~uiface() = 0;
    virtual void create_ui_element(const data&) = 0;
};
```

```
uiface::~uiface(){}

class framework
{
public:
    framework(): uiface_(0)
    {
        data dat;
        dat.url = "stdex.sf.net";
        cont_.push_back(dat);
        dat.url = "nanaproject.wordpress.com";
        cont_.push_back(dat);
    }

    void set(uiface * uif)
    {
        uiface_ = uif;
    }

    void work()
    {
        if(uiface_)
        {
            for(std::vector<data>::const_iterator i = cont_.begin();
                    i != cont_.end(); ++i)
            {
                uiface_->create_ui_element(*i);
            }
        }
    }
private:
    uiface * uiface_;
    std::vector<data> cont_;
};
```

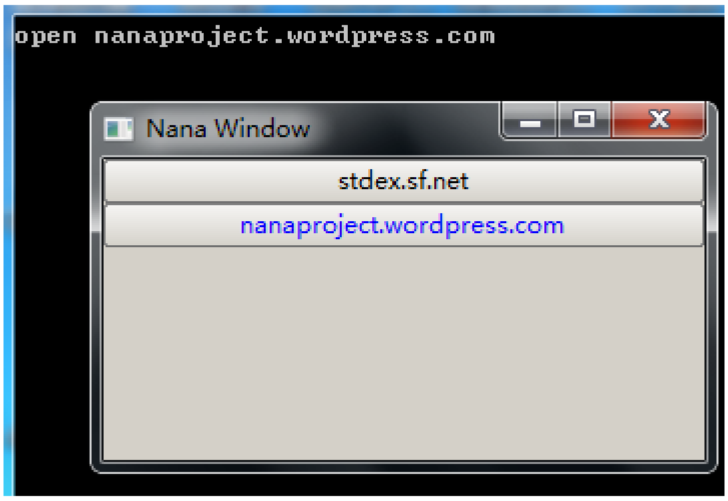
Now we need a GUI and it creates some buttons for representing the data, and when a buttons is clicked, it outputs the data that button represents. And then, let’s define a GUI for that requirement.

```
namespace ui
{
    using namespace nana::gui;
    class bar
        : public form, public uiface
    {
    public:
        bar()
        {
            gird_.bind(*this);
        }
    private:
        typedef std::pair<std::shared_ptr<button>, data> btn_pair;

        //Now we implement the virtual function that declared by uiface
        virtual void create_ui_element(const data& dat)
        {
            btn_pair p(std::shared_ptr<button>(new button(*this)), dat);
            //Make the click event
            p.first->make_event<events::click>(*this, &bar::_m_click);
            p.first->caption(nana::stringset_cast(dat.url));
            gird_.push(*(p.first), 0, 22);
            ui_el_.push_back(p);
        }
    private:
        void _m_click(const eventinfo& ei)
        {
            //Check which button is clicked
            for(std::vector<btn_pair>::iterator i = ui_el_.begin(); i !=
ui_el_.end(); ++i)
            {
                if(i->first->handle() == ei.window)
                {
                    //Show data
                    i->second;
                    std::cout<<"open "<<i->second.url<<std::endl;
                    break;
                }
            }
        }
    private:
        gird gird_;
        std::vector<btn_pair> ui_el_;
    };
}
```

It’s done for implementing the GUI. Let’s make the framework work.

```
#include <nana/gui/wvl.hpp>
#include <nana/gui/widgets/button.hpp>
#include <nana/gui/layout.hpp>
#include <iostream>

include definition of framework…

int main()
{
    ui::bar bar;
    bar.show();

    framework fw;
    fw.set(&bar);
    fw.work();

    nana::gui::exec();
}
```

Run the program. Refer to figure 2.2, when we click a button, the program would output the data that is represented by button.



```
open nanaproject.wordpress.com
```

Figure 2.2 Design of a framework

Let’s rethink the implementation of class bar. It is a bit complicated, because a check of which button is clicked is needed. In fact, we can reduce the complexity of previous design by employing function object. Think about the following implementation.

```
class bar
    : public form, public uiface
{
    struct command
    {
        data dat;
        command(const data& d): dat(d)
        {}

        void operator()()
        {
            std::cout<<"open "<<dat.url<<std::endl;
        }
    };
public:
    bar()
    {
        gird_.bind(*this);
    }
private:
    //Now we implement the virtual function that declared by uiface
    virtual void create_ui_element(const data& dat)
    {
        std::shared_ptr<button> p(new button(*this));
        //Make the click event
        p->make_event<events::click>(command(dat));
        p->caption(nana::stringset_cast(dat.url));
        gird_.push(*p, 0, 22);
        ui_el_.push_back(p);
    }
private:
    gird gird_;
    std::vector<std::shared_ptr<button> > ui_el_;
};
```

As you see it, the new implementation is more convenient. The check of which button is clicked and the pair structure are removed. In C++11, the standard library provides a bind function. Generating a function object by using std::bind() instead of giving a explicit definition of a class that used for a function. Let's remove the definition of `struct bar::command` and reimplement the `create_ui_element()`.

```
virtual void create_ui_element(const data& dat)
{
    std::shared_ptr<button> p(new button(*this));
    //Make the click event
    typedef nana::functor<void()> fun; //or std::function<void()>
    p->make_event<events::click>(
            fun(std::bind(&bar::_m_click, this, dat))
            );
    p->caption(nana::stringset_cast(dat.url));
    gird_.push(*p, 0, 22);
    ui_el_.push_back(p);
}

void _m_click(const data& dat)
{
    std::cout<<"open "<<dat.url<<std::endl;
}
```

The member function _m_click() is very tiny, we can remove it with lambda expression in C++11.

The main idea of this section is binding the execution context at some place in code and retaining the context for future use. As above code shown, we bind the data with a function object, and when the function object is called, we could easily refer to the data. By using the reasonable technical idioms, we can make the code more slighter, more expressive, the benefits are: flexible application, less comment and easy maintenance.

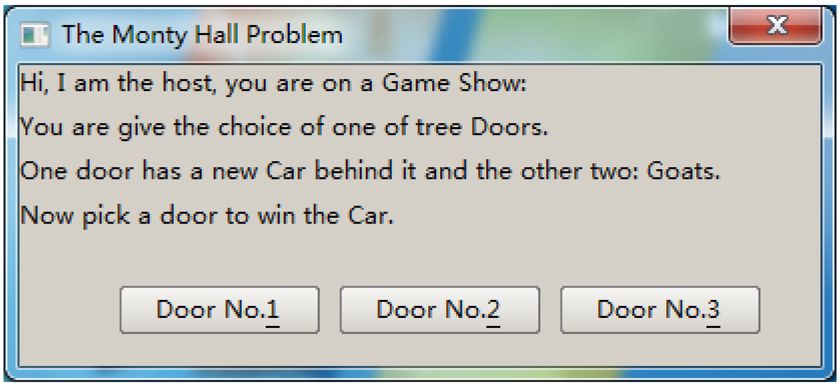
3, Creating forms

This chapter will teach you how to create a window using Nana C++ Library. A window is a visual element in operation system or a window environment. Form is a window like a container can contain other widgets, such as button, textbox and listbox. Every GUI program created by Nana C++ Library need to create one form at least, as examples in chapter 1, many forms are created by creating an instance of form, the way creating a form is easy, but it will make your program complicated if you want to create a bit larger program.

3.1, Defining a form through Derivation

Nana C++ Library is implemented using Object-oriented methods, we can define an own form by deriving from nana::gui::form.

Our first example is a Monty Hall Problem, this is a game that tests you whether change your choice after you picked a door to win a new Car.


The Monty Hall Problem

Hi, I am the host, you are on a Game Show:
You are give the choice of one of tree Doors.
One door has a new Car behind it and the other two: Goats.
Now pick a door to win the Car.

Figure 3.1 The Monty Hall Problem

Let's start creating this application. First we are going to design an architecture for the application. As we saw in figure 3.1, the application needs a form, a label and three buttons.

```
#include <nana/gui/wvl.hpp>
#include <nana/gui/widgets/label.hpp>
#include <nana/gui/widgets/button.hpp>
#include <nana/system/platform.hpp>

class monty_hall
    : public nana::gui::form
{
    enum state_t{state_begin, state_picked, state_over};
public:
    monty_hall();
private:
    void _m_pick_door(const nana::gui::eventinfo& ei);
    void _m_play(int door);
    void _m_remove_door(int exclude);
private:
    state_t state_;
    int door_has_car_;
    nana::gui::label label_;
    nana::gui::button door_[3];
};

int main()
{
    monty_hall mh;
    mh.show();
    nana::gui::exec();
26  }
```

Lines 1 to 4 include the headers that are required by this application. The wvl.hpp provides the basis of Nana C++ Library that all GUI program required, label.hpp and button.hpp include the definitions of label and button, and we will specify the door which has the new Car randomly by using timestamp(), platform.hpp provides the timestamp() function.

Lines 5 to 20 define the class monty_hall. The monty_hall is derived from nana::gui::form, in another word, class monty_hall is defined as a form and we will put all handlers in this class scope to keep program clear.

Line 8 defines a state type that contains three states, state_begin is a state that indicates the first pick, state_picked indicates the second pick, state_over indicates the result.

Line 10 declares a default constructor, it will initialize the widgets and set the initial state.

Line 12 declares a private member function to respond the user action on buttons.

Line 13 and 14 declare two private member functions which handle the game logic. We will define them later.

Lines 16 to 19 define the data members including label and buttons. The integer member door_has_car_ indicates the number of door which has a car behind it.

Lines 21 to 26 define the main function, an object of class monty_hall is defined, sets the form visible and enters the event loop.

Let’s implement the default constructor and three private member functions.

```
27  monty_hall()
28      :   nana::gui::form(nana::gui::API::make_center(400, 150),
29                                  appear::decorate<appear::taskbar>())
30          ,state_(state_begin)
31  {
32      this->caption(STR("The Monty Hall Problem"));
33      nana::string text = STR("Hi, I am the host, you are on a Game Show:\n")
34          STR("You are given the choice of one of tree Doors.\n")
35          STR("One door has a new Car behind it and the other two: Goats.\n")
36          STR("Now pick a door to win the Car.");
37
38      label_.create(*this, 0, 0, 400, 100);
39      label_.caption(text);
40
41      nana::string door_name[3] =
42                  {STR("Door No.&1"), STR("Door No.&2"), STR("Door No.&3")};
43      for(int i = 0; i < 3; ++i)
44      {
45          door_[i].create(*this, 50 + 110 * i, 110, 100, 24);
46          door_[i].caption(door_name[i]);
47          door_[i].make_event<nana::gui::events::click>(*this,
48                                              &monty_hall::_m_pick_door);
49      }
50  }
```

Line 28 and 29 initialize the base class of monty_hall, make_center() is a function returns a rectangle that specifies an area in the center of screen with size of 400X150. The typedef name appear is used for the abstraction of form appearance, the appear::decorate defines the form with caption bar, border, a close button and displaying in taskbar.

Line 30 initializes the initial state.

Line 32 sets the caption of form.

Lines 33 to 36 define a text that describes the game information.

Line 38 and 39 create a label widget and set the caption of label. The label will display the text which is defined in line 33.

Line 41 and 42 define a string array which contains the names of three doors.

Lines 43 to 48 create the buttons in a loop, and set the caption, make a click event for three buttons.

```
51  void _m_pick_door(const nana::gui::eventinfo& ei)
52  {
53      int index = 0;
54      for(; index < 3; ++index)
55      {
56          if(door_[index] == ei.window)
57              break;
58      }
59      _m_play(index);
    }
```

On line 51, the member function has a parameter, eventinfo contains the event information, such as which widget the event is taking place. We need the information to determinate which button is clicked, because the event handler of three buttons are one and same.

Lines 53 to 59 are finding the index of the button which is clicked. The button which is click is specified by ei.window. _m_play() handles the logic of game.

```
60  void _m_play(int door)
61  {
62      switch(state_)
63      {
64      case state_begin:
65          door_has_car_ = (nana::system::timestamp() / 1000) % 3;
66          _m_remove_door(door);
67          state_ = state_picked;
68          break;
69      case state_picked:
70          label_.caption(door_has_car_ == door ?
71              STR("Yes, you win the new Car!!") : STR("Sign, you are lost!"));
72          state_ = state_over;
73          break;
        }
    }
```

Line 60 defines the _m_play() to handle the logic of this game, it contains a parameter to indicate what the number of door is picked. There are two states we would handle, state_begin indicates the first pick, and now, program should put the car behind a door randomly, use the timestamp() function to retrieve the number of milliseconds that have elapsed since system was started, we use the number to implement randomization, and find the remainder of division of the number by 3 through the modulus operation, the last result is the door number which is large or equal to 0 and less than 3. _m_remove_door() is used for removing a door excluding the car behind it and the door that is picked.

Line 67 is setting the next state.

Line 69 is start of handling the state that gamer plays the second pick.

Line 70 and 71 test the number of door whether is the door which the car behind it, and set the caption of label by result.

Line 72 sets the final state.

In fact, in the lines from 51 to 59, the member function _m_pick_door() can be removed by using std::bind(), refer to section **2.3 Think about the Design**. By using std::bind(), we can bind the the index of door to the _m_play() and make it as the event handler for the click of button.

```
74  void _m_remove_door(int exclude)
75  {
76      std::vector<int> doors;
77      for(int i = 0; i < 3; ++i)
78      {
79          if(i != exclude)
80              doors.push_back(i);
        }
81      unsigned ts = (nana::system::timestamp() / 1000) % 2;
82      if(door_has_car_ == doors[ts])
83          ts = (ts ? 0: 1);
84      door_[doors[ts]].enabled(false);
85      doors.erase(doors.begin() + ts);

86      nana::string text = STR("I know what's behind all the doors and")
87              STR("I remove a door which a goat behind it. \n")
88              STR("And now, do you want to stick with your decision")
89              STR(" of Door No.X or do you want to change your choice")
90              STR(" to Door No.Y?");
91      nana::char_t door_char = '1' + exclude;
92      nana::string::size_type pos = text.find(STR("Door No.X"));
93      text.replace(pos + 8, 1, 1, door_char);

94      door_char = '1' + doors[0];
95      pos = text.find(STR("Door No.Y"));
96      text.replace(pos + 8, 1, 1, door_char);
97      label_.caption(text);
    }
```

Line 74 defines the _m_remove_door() to removes a door which is a goat behind it and is not picked by gamer. The parameter exclude is a door number that picked by gamer.

Lines 76 to 80 create a std::vector object that contains the index of doors excluding the one of gamer picked. So the result of vector only contains two doors.

Lines 81 to 85 choose a door in the vector randomly, if the door has a car behind it, change the choice to other one, the door which is chosen will be removed. A widget has two methods named enabled(), they are overloading functions `void enabled(bool)` and `bool enabled() const`, are used for setting or retrieving the state whether the widget can receive inputs from mouse and keyboard. If the enabled state of a button is set to false, the color of button text becomes gray and the button is not working while clicking on it.

Lines 86 to 97 prepare the text that asks gamer whether changes his choice after the first pick.

Now run the program. Verify that the shortcut keys Alt+1, Alt+2 and Alt+3 trigger the correct behavior. Press Tab key to navigate through the buttons with keyboard, the default tabstop order is the order in which the buttons were created.

3.2, Appearance of Window

A window has an appearance, the appearance can be specified when a window is being created. There is a structure named appearance to determine the appearance of a window. The definition of the structure is

```
struct appearance
{
    bool taskbar;
    bool floating;
    bool no_activate;
    bool minimize;
    bool maximize;
    bool sizable;
    bool decoration;
};
```

In practical development, the struct appearance is hard to describe what style of the appearance is, so Nana provides three templates class to generate an appearance object for readability and understandability.

```
struct appear
{
    struct minimize;
    struct maximize;
    struct sizable;
    struct taskbar;
    struct floating;
    struct no_activate;

    template<    typename Minimize = null_type,
                 typename Maximize = null_type,
                typename Sizable = null_type,
                typename Floating = null_type,
                typename NoActive = null_type> struct decorate;

    template<    typename Taskbar = null_type,
                typename Floating = null_type,
                typename NoActive = null_type,
                typename Minimize = null_type,
                typename Maximize = null_type,
                typename Sizable = null_type> struct bald;

    template<    bool HasDecoration = true,
                typename Taskbar = null_type,
                typename Floating = null_type,
                typename NoActive = null_type> struct optional;
};
```

These templates generate appearances, and every template receives the template parameters for specifying the attributes of appearance.

template decorate is used for generating an appearance with a border and titlebar.

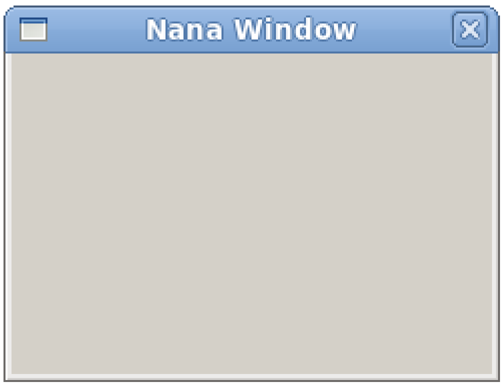


Figure 3.2 Decoration

The appearance of Figure 3.2 that created by appear::decorate<>() has a titlebar and borders that are draw by platform window manager.

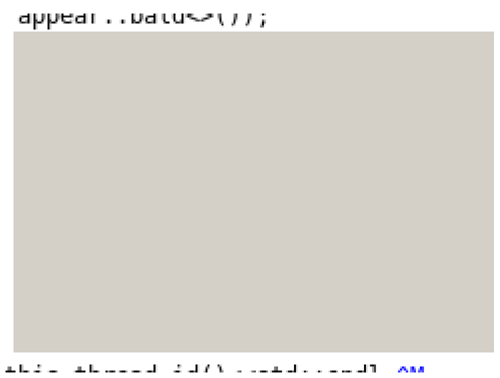


Figure 3.3 No decoration

The appearance of Figure 3.3 that created by appear::bald<>() has not a titlebar and 3D-look borders.

The appearances of Figure 3.2 and 3.3 are created by templates without specifying the template parameters. Each template provides some parameters, and each template parameter name indicates what attribute is supported by this template. Such as, template decorate, it receives the parameters are appear::minimize, appear::maximize, appear::sizable, appear::floating and appear::no_activate. For example, creating a appearance with decoration that has a minimize button and maximize button. See code blow.

```
#include <nana/gui/wvl.hpp>

int main()
{
    using namespace nana::gui;
    form fm(API::make_center(240, 160),
            appear::decorate<appear::minimize, appear::maximize>());
    fm.show();
    exec();
}
```

The appearance will be displayed like Figure 3.4

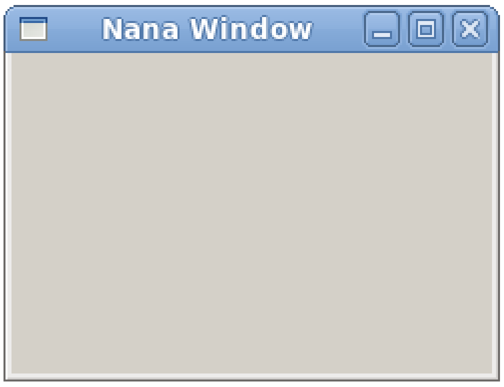


Figure 3.4 Decoration with minimize button and maximize button

There is a feature for these templates, the order of template parameters does not affect the order of explicit specifying template parameters, the name of template parameters just tell you which attribute can affect the appearance. For example.
appear::decorate<appear::minimize, appear::maximize>() creates an appearance that is the same as the result that created by appear::decorate<appear::maximize, appear::minimize>().
appear::decorate<>() creates an appearance that is the same as appear::decorate<int, int, int, int, int>(), because the type of int is not wanted.

3.3, Creating An Instance of the Form

In the previous chapters we provided examples of defining the form objects, they are usually local objects. When the local object goes out of scope, its destructor is called automatically. Sometimes we need to keep the object alive when it goes out of the scope, another way to create an object is to declare a pointer variable to the class of object and call the C++ new operator, which will allocate space for the object and call the constructor. In this case, the pointer variable must be explicitly deallocated with the delete operator. But the question is when shall we delete the pointer variable? In fact, we can create a form object by using form_loader. Nana C++ Library will manage the form objects that created by form_loader and destroy the objects when the user close the form. For example.

```
#include <nana/gui/wvl.hpp>

int main()
{
    using namespace nana::gui;
    form_loader<form>()().show();
    exec();
}
```

nana::gui::form_loader is a template functor class, it creates an object of the template parameter class. nana::gui::form_loader is useful when you create a form and don't manage to take care about the lifetime of the object. Continuing with the the next example, we see a form is created when the button is being clicked.

```
#include <nana/gui/wvl.hpp>
#include <nana/gui/widgets/button.hpp>

void click()
{
    using namespace nana::gui;
    form_loader<form>()().show();
}

int main()
{
    using namespace nana::gui;

    form fm;
    button btn(fm, 10, 10, 150, 23);
    btn.caption(STR("Open a new form"));
    btn.make_event<events::click>(click);
    fm.show();
    exec();
}
```

In fact, a form_loader object can be passed to make_event because it is a functor. Take a look at above code, form would not be shown immediately after initialization, and the show() method is need to call explicitly after creating the form by form_loader. Therefore, only an invisible form that created by the form_loader object we can get, if we pass the form_loader object like above code to the make_event. A way to achieve it is calling the show() method in that constructor of class, but in most instances, we are not enable to modify the code of the classes, and therefore we need a correct solution. The form_loader has two template parameters, the first is used for specifying a type of form, the second is a non-type template parameter and its type is bool named IsMakeVisible, used for determining whether making the form visible. So, we can specify the second template parameter by giving true, like this.

```
btn.make_event<events::click>(form_loader<form, true>());
```

3.4, Modal Form

Modal form is a modality of forms. It would handle all interactions with the user while the form is active. This is what makes the form modal, the user can not interact with its owner window until the form is closed. Modal form is useful to block the program execution and wait for a user input. For example.

```
#include <nana/gui/wvl.hpp>
#include <nana/gui/widgets/button.hpp>
#include <iostream>

void foo(const nana::gui::eventinfo& ei)
{
    using namespace nana::gui;
    form fm(ei.window, API::make_center(ei.window, 400, 300));
    fobj.caption(STR("I am a modal form"));

    std::cout<<"Block execution till modal form is closed"<<std::endl;
    API::modal_window(fm);
    std::cout<<"modal form is closed"<<std::endl;
}

int main()
{
    using namespace nana::gui;

    form fm;
    fm.caption(STR("Click me to open a modal form"));
    fm.make_event<events::click>(foo);
    fm.show();
    exec();
}
```

Call nana::gui::API::modal_window() to enable modal form. Only if an owner is specified for the form initialization, will the form enable as modal form. The object fm in function foo() is created and specified ei.window as its owner. The ei.window refers to the form that is defined in function main(), and passed by event argument.

4, Event Handling

Event is a messaging mechanism and provided by window system, various kinds of events are generated by window system for notifying the application. A key or mouse event is generated when the user presses or releases a key or mouse button, and the application can receive the event and response to user actions.

The different window systems provide their own patterns of messaging mechanism. Nana implements an abstract pattern of event mechanism to hide the difference between different window systems.

4.1, Registering and Unregistering an Event

To response to the click action that generated by any specified button, we need to make the click event for the button.

```
#include <nana/gui/wvl.hpp>
```

```
#include <nana/gui/widgets/button.hpp>

void foo()
{
}

int main()
{
    using namespace nana::gui;
    form fm;
    fm.make_event<events::click>(foo);
    exec();
}
```

As shown in above code, the make_event() is a member template of class widget, and it is used for registering an event handler. The above code makes a click event for the form object, when the user clicks the body of the form, Nana is responsible for invoking the function foo() that registered as a handler of the specified event. Additionally, the function foo() can be specified with a parameter, that type is const nana::gui::eventinfo& to receive the information of the event. The details of eventinfo are described in section 4.2.

The make_event() returns a handle of event handler, if the registration is successful. The type of the return value is nana::gui::event_handle. With the handle, we can delete the event handler manually. For example.

```
event_handle handle = fm.make_event<events::click>(foo);
fm.umake_event(handle);
```

After calling the umake_event(), the foo() function would be not invoked when the user clicks the body of form. In most situation, we could not take care of deleting the event handler, because the event handlers will be deleted while form is closed.

4.2, Events

An event type describes a specific event which is generated by Nana. For each event type, a corresponding class is defined by Nana, which is used for referring to an event type. The classes for event types are defined in the namespace nana::gui::events, such as click, mouse_move and so on.

Every event contains some information, the information is sent to the application through a reference of eventinfo object in event handler.

```
#include <nana/gui/wvl.hpp>
void foo(const nana::gui::eventinfo& ei)
{
    //Refer to ei for information of event.
}

int main()
{
    using namespace nana::gui;
    form fm;
    fm.make_event<events::click>(foo);
    fm.show();
    exec();
}
```

For different events, the eventinfo contains different structures for these event types. The following paragraphs describe the details of eventinfo.

4.2.1, Mouse Events

The mouse is an important and convenient input device. The user controls the mouse cursor on the screen through moving and clicking mouse. If the mouse cursor moves over on a widget, the system would generate a mouse_move event to notify the program to which the widget belongs.

The structure of mouse events contains:

```
struct implemented-specified
{
    short x;
    short y;              //point (x, y) coordinates in event widget
    bool left_button;     //True if the left button is pressed.
    bool mid_button;      //True if the middle button is pressed.
    bool right_button;    //True if the right button is pressed.
    bool shift;           //True if the shift key is pressed.
    bool ctrl;            //True if the ctrl key is pressed.
};
```

The structure of mouse wheel event contains:

```
struct implemented-specified
{
    short x;
    short y;          //point(x, y) coordinates in event widgets.
    bool upwards;    //True if the wheel is upward rolls.
    bool shift;      //True if the shift key is pressed
    bool ctrl;       //True if ctrl key is pressed.
};
```

In an event handler function, we can refer to the structure by using eventinfo object. For example:

```
void foo(const nana::gui::eventinfo& ei)
{
    using namespace nana::gui;
    switch(ei.identifier)
    {
    default:
        ei.mouse;    //refers to the structure of mouse events.
        break;
    case events::mouse_wheel::identifier:
        ei.wheel;    //refers to the structure of mouse_wheel event.
        break;
    }
}
```

To receive a mouse event, register the event with a specific class which is defined in namespace nana::gui::events.

click/dbl_click
When the user clicks mouse and the mouse cursor is in a widget, the widget receives a click event that is generated by Nana.

mouse_enter/mouse_move/mouse_leave
This event is sent to a widget when the mouse cursor enters/leaves the widget.

mouse_down/mouse_up
This event is sent when the user presses/releases/moves the mouse while the mouse cursor is in a widget.

4.2.2, Keyboard Events

There are three different kinds of keyboard events in Nana C++ Library: key down, key up and character key down.

A window system usually uses an input focus to represent a window which would receive the keyboard events. In general, a window which user clicks would be set to the input focus widow in Nana C++ Library, additionally, the program can determinate which window gets the input focus with calling nana::gui::API::set_focus().

The structure of keyboard event contains:

```
struct implemented-specified
{
    mutable nana::char_t key;
    mutable nana::bool ignore;
    unsigned char ctrl;
};
```

As the definition shown, the member key and ignore are defined as mutable, the feature will be explained later.

key_down/key_up
When the users hit the keyboard, the mouse_down would be generated when the key is pressed down, and the mouse_up would be generated when the pressed key is released.

key_char
The key_char event is an abstract event. A window system usually translates the keys into characters. For example, to type a Chinese character usually needs to hit the keyboard more than one key, the window system translates these keys into a Chinese character and then a key_char event is generated and sent to the program.

The two members, key and ignore, are defined as mutable in the structure of key event, is used for modifying the state of key_char event. During key_char is processing, if the member "ignore" is set to true, the Nana will igore the key. For example, when a program is designed to receive the number input, the program should test the key in key_char event, and set the "ignore" to true if the input char is not a digital. Like blow code.

```
void only_digital_allowed(const nana::gui::eventinfo& ei)
{
    ei.ignore = (ei.key < '0' || ei.key > '9');
}
```