# Lecture 21

Password: friedTofu

## Graphical User Interface

- Introduction
- Available toolkits and problems
- Structure of a GUI program
- How to make a (G)UI toolkit
- Canvas, Menu, Dialog, and Widget

## Introduction

- Graphical User Interface (GUI) was invented in Xerox lab, and made popular by Apple. (Everyone knows the legend of Steve Jobs)
- One of the major obstacles for cross-platform development.
- Also it is one of the most common reasons why the code rots.

## Why GUI?

- Why GUI?  Aren’t FsSimpleWindow and FsLazyWindow frameworks good enough for testing algorithms?
- When you are working alone, that’s probably good enough.
- But, in many situations, you work with someone else, and you need to share your program with your colleagues.
- Even by yourself, you may run out of keys for testing features.  Or, can you remember all keys?
- When the scale of your program grows beyond certain point, it might be more efficient to spend some time to set up your GUI than using a primitive user interface.

## Available toolkits and problems

- MFC (Microsoft Foundation Classes)
  - Large legacy code base.
  - Runs only on Windows platforms.
  - Almost impossible to programmatically create dialog boxes without the Resource Editor in the Visual Studio.

- Document-View Architecture
  - The application class (called theApp) manages documents (sub-class of CDocument class) and views (sub-class of CView class).
  - A document is a abstract data (core data) structure that the application is going to create and edit.
  - A view is almost same as a window.

## Available toolkits and problems

- Win32 API
  - Base API for MFC.
  - A low-level interface functions that also can manage GUI widgets.
  - Runs only on Windows platforms.
  - Easier to programmatically create dialog boxes without the Resource Editor, but not very easy.
- Every widget is a window.
- An event is sent to a function called window procedure, which is associated to a window.
- An appropriate call-back function needs to be called from the window procedure -> The window procedure becomes huge when the scale of the application grows.

## Available toolkits and problems

- Cocoa
  - API for Mac OSX and iOS.
  - Requires Objective-C or Swift.
  - Not very easy to programmatically create dialog boxes.
- Objective-C is a problem.  Due to fragile language foundation, it went through ad-hoc change of language specification, which forced programmers to re-write existing code.

## Available toolkits and problems

- Tcl/Tk
  - Open source programming language.
  - Once was very popular.
  - Does not really make sense to learn a separate programming language only for building GUI.
  - Losing popularity when Qt became popular.

- Question can be: Do you want to learn a whole new programming language only to build a user interface? (I don’t.)

## Available toolkits and problems

- Qt
  - Cross-platform and open source user-interface toolkit.
  - Somewhat popular.
  - Not too difficult to create dialog boxes programmatically.
  - Developed by Nokia, but Nokia ditched the project (and then bought by Microsoft.)
  - Now maintained as an open-source project.
  - Too much DLLs to ship with.
  - Questionable future:

    Is an open source project automatically future proof?  I don’t think so. An open-source project may collapse when the source code gets obese and unmaintainable.  The biggest problem is that nobody takes responsibility of an open source project.

    If the code is beyond certain scale, it is nothing future proof.

## The MAJOR problem

- Once Win32/MFC. Then Tcl/Tk. Now Qt. What's next?
- We are forced to re-write GUI part every time something new gains popularity.
- Absolute waste of time because we need to learn something new to do the same thing. (Not to do the same thing better. Just to do exactly the same thing.)

- (Have you heard about 'penance of sand painting'?)

- If you need to write a GUI code in your work, probably you will be asked to write some Qt code.
- But, don’t be surprised if one day your boss comes to you, and tell “Sorry, Qt is over.  Re-write whole your GUI code with this toolkit called XYZ”

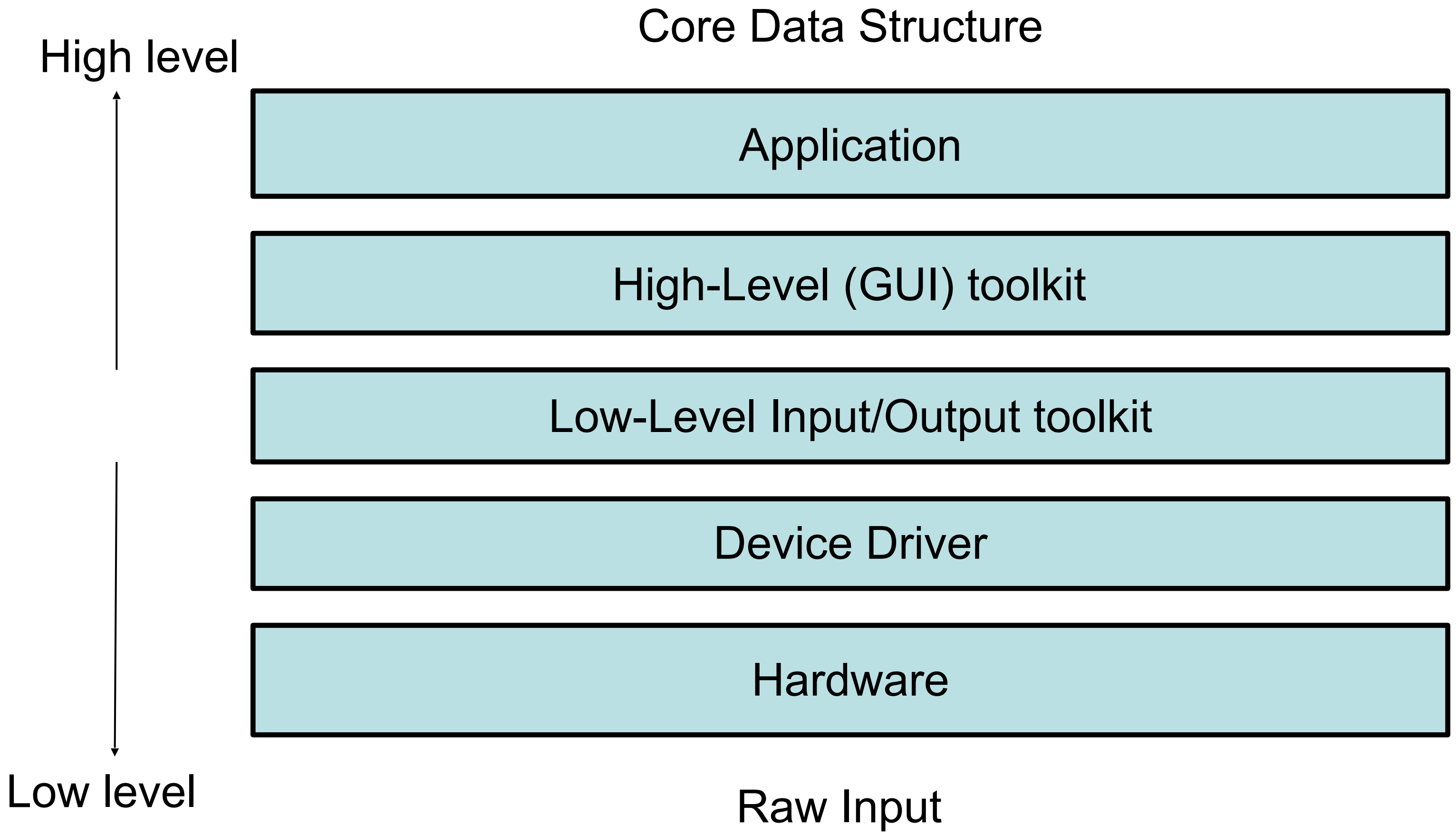
## Solution?

- Write core data structure and algorithms of your program independent of the GUI as much as possible.
- Make it portable across different GUI toolkits.

- Learn how such toolkits work.
- Be prepared to write your own if it is necessary.

## Structure of a GUI program

- High-Level GUI API lies between Low-Level Input/Output API and Application.



## Structure of a GUI program

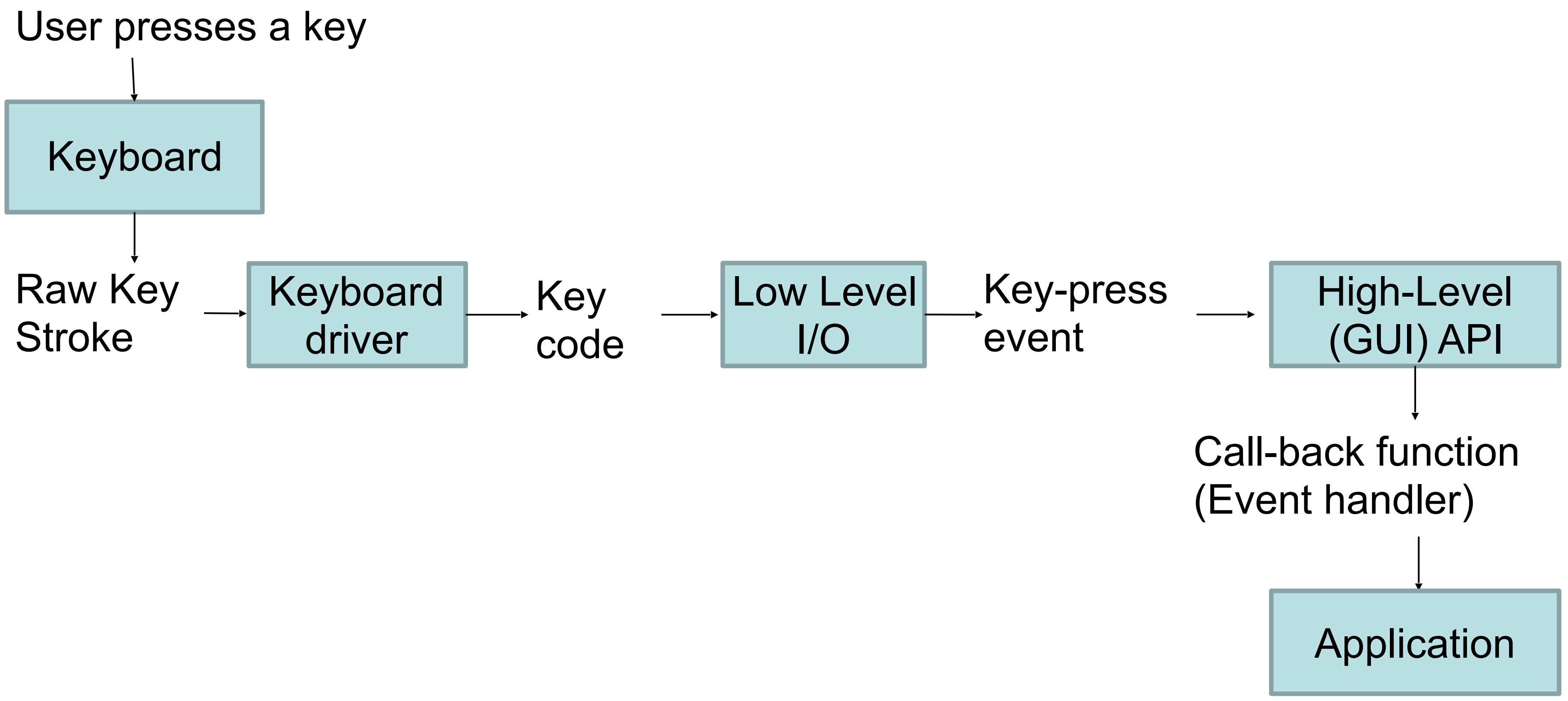
- High-Level (GUI) toolkit calls a call-back function (called event handler) of the Application based on an event.
- The role of the GUI toolkit is to interpret an event from the low-level toolkit and calls an appropriate event handler.
- Example of Raw Events:
  - Key stroke
  - Mouse button
  - Mouse move
  - Timer
  - Drag & Drop
  - Window exposure
  - Window resizing
  - Touch
  - Etc.

## Structure of a GUI program

- High-level events may be derived from a raw event.
- Example of Derived Events:
  - Menu selection
  - Button press
  - Dialog open
  - Dialog closure
  - Etc.

## Structure of a GUI program

- Example: Key stroke



## Structure of a GUI program

- A GUI toolkit typically consists of:
  - A set of libraries.
  - An application template (template project)
- An application can be created by modifying the template.
- The template must be small.
  - The most of the functionalities must be in the library side.
  - An improvement of the library should also improve the application derived from the template.
- It should be possible to derive another application from an existing application.
  - Example: Creating an mesh-editor from a mesh-viewer.

## Structure of a GUI program

- Deriving a new application from an existing application.
  - Easy to make a copy and continue.  But,…
  - Let’s say you derive MeshEditor program from MeshViewer.
  - You make a copy of MeshViewer and then expanded it to be a MeshEditor.
  - Later you added features to MeshViewer.
  - How can you import the new features to MeshEditor?
  - Manually merging the new MeshViewer into MeshEditor could be very tedious.

- The GUI framework and application should be set for this situation.

## Strategy of making a GUI toolkit

General strategy:

- Make a base class of the application.
- An application creates a sub-class of the application base class.

Question: How event-handlers should be implemented?

- ???
- ???
- ???

## Example by virtual functions

- Use FsLazyWindow framework as a low-level I/O toolkit.
- Write an application base class.
  - AddMenu function. Can be used from the sub-class (actual application)
  - DrawGui function. Should be
  - OnInitialize function. A virtual function called at the beginning.
  - OnDraw function. A virtual function called when the window needs to be drawn.
  - OnKeyDown function.
- Create a sample application that:
  - R-key to change the background color to red.
  - G-key to change the background color to green.
  - B-key to change the background color to blue.
- Key strokes are not really GUI, but let’s pretend a key stroke is a GUI event.

## Example by virtual functions

- main.cpp
- guibase.cpp
- guibase.h

$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ Application base classes

- application.h
- application.cpp
- singleton.cpp

$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ Application classes (implementation)

(*) Singleton is an object that only one object exists in one program. An application object is often a singleton. It implies (but not necessarily) to be created upfront, and stays until (or very close to) the termination of the program.

## main.cpp

- From public/src/fslazywindow/template

```
void FsLazyWindowApplication::Initialize(int argc,char *argv[])
{
    YsGLSLRenderer::CreateSharedRenderer();
    GuiBase::GetGuiApplication()->OnInitialize();
}

void FsLazyWindowApplication::Interval(void)
{
    auto key=FsInkey();
    if(FSKEY_NULL!=key)
    {
        GuiBase::GetGuiApplication()->OnKeyDown(key);
    }
    if(FSKEY_ESC==key)
    {
        SetMustTerminate(true);
    }
    needRedraw=true;
}

void FsLazyWindowApplication::Draw(void)
{
    GuiBase::GetGuiApplication()->OnDraw();
    needRedraw=false;
}
```

## guibase.h

```
#ifndef GUIBASE_IS_INCLUDED
#define GUIBASE_IS_INCLUDED

#include <vector>
#include <string>

class GuiBase
{
public:
    class MenuItem
    {
    public:
        std::string msg;
    };
private:
    std::vector <MenuItem> menu;

public:
    virtual ~GuiBase();

    void AddMenu(const std::string &msg);
    void DrawGui(void) const;

    virtual void OnInitialize(void);
    virtual void OnDraw(void);
    virtual void OnKeyDown(int fskey);

    static GuiBase *GetGuiApplication(void);
};

#endif
```

Event handlers

# guibase.cpp

```
#include <ysclass.h>
#include <ysgl.h>
#include <ysglcpp.h>
#include <ysglslcpp.h>
#include <fslazywindow.h>

#include "guibase.h"

GuiBase::~GuiBase()
{
}

void GuiBase::AddMenu(const std::string &msg)
{
    MenuItem newMenu;
    newMenu.msg=msg;
    menu.push_back(newMenu);
}
```

# guibase.cpp (continued)

```
void GuiBase::DrawGui(void) const
{
    YsGLSLBitmapFontRendererClass renderer;
    int viewport[4];
    glGetIntegerv(GL_VIEWPORT,viewport);

    renderer.SetViewportDimension(viewport[2],viewport[3]);
    renderer.RequestFontSize(16,24);

    GLfloat col[4]={0,0,0,1};
    renderer.SetUniformColor(col);

    int y=24;
    for(auto &m : menu)
    {
        renderer.DrawString(0,y,m.msg.c_str());
        y+=24;
    }
}

void GuiBase::OnInitialize(void)
{
}
void GuiBase::OnDraw(void)
{
}
void GuiBase::OnKeyDown(int fskey)
{
}
```

Default behavior: Do nothing

## singleton.cpp

```
#include "application.h"

static Application *appPtr=nullptr;

GuiBase *GuiBase::GetGuiApplication(void)
{
    if(nullptr==appPtr)
    {
        appPtr=new Application;
    }
    return appPtr;
}
```

The purpose of singleton.cpp is to create and store one instance of Application-class object.

## application.h

```
#ifndef APPLICATION_IS_INCLUDED
#define APPLICATION_IS_INCLUDED

#include "guibase.h"

class Application : public GuiBase
{
public:
    float bgColor[4];

    virtual void OnInitialize(void);
    virtual void OnDraw(void);
    virtual void OnKeyDown(int fskey);

    void SetRedBackground(void);
    void SetGreenBackground(void);
    void SetBlueBackground(void);
};

#endif
```

Behaviors of these event handlers must be implemented in this class

Actual event handlers for the key strokes.

# application.cpp

```
#include <ysclass.h>
#include <ysgl.h>
#include <ysglcpp.h>
#include <ysglslcpp.h>
#include <fslazywindow.h>

#include "application.h"

void Application::OnInitialize(void)
{
    bgColor[0]=1;
    bgColor[1]=1;
    bgColor[2]=1;
    bgColor[3]=1;
    AddMenu("R-key  Red");
    AddMenu("G-key  Green");
    AddMenu("B-key  Blue");
}
void Application::OnDraw(void)
{
    glClearColor(bgColor[0],bgColor[1],bgColor[2],bgColor[3]);
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    DrawGui();
    FsSwapBuffers();
}
```

# application.cpp (continued)

```
void Application::OnKeyDown(int fskey)
{
   switch(fskey)
   {
   case FSKEY_R:
      SetRedBackground();
      break;
   case FSKEY_G:
      SetGreenBackground();
      break;
   case FSKEY_B:
      SetBlueBackground();
      break;
   }
}

void Application::SetRedBackground(void)
{
   bgColor[0]=1;   bgColor[1]=0;   bgColor[2]=0;
}
void Application::SetGreenBackground(void)
{
   bgColor[0]=0;   bgColor[1]=1;   bgColor[2]=0;
}
void Application::SetBlueBackground(void)
{
   bgColor[0]=0;   bgColor[1]=0;   bgColor[2]=1;
}
```

Redirect to the actual event-handlers for the key strokes.

## Virtual-function approach

- Is this good?  Are we happy with this approach?

## Example by virtual functions

Problem

- Event routing is a responsibility of the application.
- Menu addition and event routing are separate.
- When you change the key assignment, you also need to change routing (in OnKeyDown) consistently.

- Not ideal. Better to be able to specify a menu-appearance and corresponding call-back function in the same location.

## Example by function pointers

- How about using a function pointer?
- Let’s assign a function pointer for each menu item, and call it from GuiBase::OnKeyDown

No changes in:

- singleton.cpp
- main.cpp
- application.h

# guibase.h

```
#ifndef GUIBASE_IS_INCLUDED
#define GUIBASE_IS_INCLUDED

#include <vector>
#include <string>
class GuiBase
{
public:
    class MenuItem
    {
    public:
        std::string msg;
        int fskey;
        GuiBase *owner;
        void (GuiBase::*func)(void);
    };
private:
    std::vector <MenuItem> menu;

public:
    virtual ~GuiBase();

    void AddMenu(const std::string &msg,int fskey,void (GuiBase::*func)(void));
    void DrawGui(void) const;

    virtual void OnInitialize(void);
    virtual void OnDraw(void);
    virtual void OnKeyDown(int fskey);

    static GuiBase *GetGuiApplication(void);
};
#endif
```

Each menu should remember which member function of which instance must be called.

AddMenu gets a little more complex than the last example

# guibase.cpp

## Changes

```
void GuiBase::AddMenu(const std::string &msg,int fskey,void (GuiBase::*func)(void))
{
    MenuItem newMenu;
    newMenu.msg=msg;
    newMenu.owner=this;
    newMenu.func=func;
    newMenu.fskey=fskey;
    menu.push_back(newMenu);
}

void GuiBase::OnKeyDown(int fskey)
{
    for(auto m : menu)
    {
        if(m.fskey==fskey)
        {
            ((m.owner)->*(m.func))();
        }
    }
}
```

Default behavior is to call the call-back function directly.

## application.cpp

### Changes

By uncommenting these three lines, a call-back function should be assigned to each key stroke.  Or not?

```
void Application::OnInitialize(void)
{
    bgColor[0]=1;
    bgColor[1]=1;
    bgColor[2]=1;
    bgColor[3]=1;

    // This is the intension, but give an error if you uncomment below.
    // AddMenu("R-key  Red"  ,FSKEY_R,&Application::SetRedBackground);
    // AddMenu("G-key  Green",FSKEY_G,&Application::SetGreenBackground);
    // AddMenu("B-key  Blue", FSKEY_B,&Application::SetBlueBackground);
}


void Application::OnKeyDown(int fskey)
{
    GuiBase::OnKeyDown(fskey);
}
```

Default behavior for a key stroke is just let the OnKeyDown function of the base class redirect to the event handler functions.

## application.cpp

### Changes

By uncommenting these three lines, a call-back function should be assigned to each key stroke.  Or not?

```
void Application::OnInitialize(void)
{
    bgColor[0]=1;
    bgColor[1]=1;
    bgColor[2]=1;
    bgColor[3]=1;

    // This is the intension, but give an error if you uncomment below.
    // AddMenu("R-key  Red",FSKEY_R,&Application::SetRedBackground);
    // AddMenu("G-key  Green",FSKEY_G,&Application::SetGreenBackground);
    // AddMenu("B-key  Blue", FSKEY_B,&Application::SetBlueBackground);
}
```

FAIL!

```
void Application::OnKeyDown(int fskey)
{
    GuiBase::OnKeyDown(fskey);
}
```

Default behavior for a key stroke is just let the OnKeyDown function of the base class redirect to the event handler functions.

## Example by function pointers

- Problem: Can only assign a function of the base class. What we want is to assign a function of the sub-class to a menu item.

- Using template somewhat solves the problem. But, it does not solve expansion problem.
- Imagine you write a mesh-viewer application, and then expand it to make a mesh-editor application.
- Menu item with the owner-class as a template parameter can only call a function of the mesh-viewer class.
- Cannot assign a function of the mesh-editor class.

## Example of function pointers

- Use a pointer to a static function.

## guibase.h

### Changes

```
class MenuItem
{
public:
    std::string msg;
    int fskey;
    void *owner;
    void (*func)(void *contextPtr);
};
```

Make it callable any function with any parameter

func can be any function that takes a void pointer as a parameter.

# guibase.cpp

## Changes

```
void GuiBase::AddMenu(const std::string &msg,int fskey,void (*func)(void *),void *contextPtr)
{
    MenuItem newMenu;
    newMenu.msg=msg;
    newMenu.owner=contextPtr;
    newMenu.func=func;
    newMenu.fskey=fskey;
    menu.push_back(newMenu);
}

void GuiBase::OnKeyDown(int fskey)
{
    for(auto m : menu)
    {
        if(m.fskey==fskey)
        {
            (m.func)(m.owner);
        }
    }
}
```

# application.h

## Changes

```
class Application : public GuiBase
{
public:
    float bgColor[4];

    virtual void OnInitialize(void);
    virtual void OnDraw(void);
    virtual void OnKeyDown(int fskey);

    static void SetRedBackground(void *contextPtr);
    static void SetGreenBackground(void *contextPtr);
    static void SetBlueBackground(void *contextPtr);
};
```

These event handlers are static functions, which does not have an associated this pointer.

contextPtr should be this pointer.

# application.cpp

## Changes

Event handlers are directly connected to key strokes. *this* pointer will be given as contextPtr for the event handlers.

```
void Application::OnInitialize(void)
{
    bgColor[0]=1;
    bgColor[1]=1;
    bgColor[2]=1;
    bgColor[3]=1;

    AddMenu("R-key  Red"  ,FSKEY_R,SetRedBackground,this);
    AddMenu("G-key  Green",FSKEY_G,SetGreenBackground,this);
    AddMenu("B-key  Blue", FSKEY_B,SetBlueBackground,this);
}
void Application::SetRedBackground(void *contextPtr)
{
    auto thisPtr=(Application *)contextPtr;
    thisPtr->bgColor[0]=1;    thisPtr->bgColor[1]=0;    thisPtr->bgColor[2]=0;
}
void Application::SetGreenBackground(void *contextPtr)
{
    auto thisPtr=(Application *)contextPtr;
    thisPtr->bgColor[0]=0;    thisPtr->bgColor[1]=1;    thisPtr->bgColor[2]=0;
}
void Application::SetBlueBackground(void *contextPtr)
{
    auto thisPtr=(Application *)contextPtr;
    thisPtr->bgColor[0]=0;    thisPtr->bgColor[1]=0;    thisPtr->bgColor[2]=1;
}
```

## Example by static function pointers

- It does get the job done.
- This has long been a work-around.

## Example of function pointers

- It does work, but….
- Problem of a pointer to a static function: I need to static-cast the first parameter (owner pointer) to the Application pointer.
- This cast needs to be done in each event handler. Static-cast everywhere!
- One typographical error buried in hundreds of similar expressions is incredibly difficult to catch.
- Want C++ to catch such an error.
- Since Application is a polymorphic sub-class of GuiBase, should use dynamic_cast for safety, but cannot.

## Does C++ have a solution?

- There was no solution before C++11.
- That’s why MFC had its own language extension for C++. Qt designer generates a C++ code from its GUI layout data and you are not supposed to modify auto-generated C++ code.
- Such auto-generated parts eventually rot and the program become unmaintainable.

## Solution finally came in C++11

- Functional object.
- You can *bind* a function call with a set of parameters.
- The differences between a function pointer and a functional object are:
  - A function pointer is just a function, while functional object can be a function + a set of parameters.
  - The type of the function pointer is rigid or stiff, while functional object is more flexible.

## How is it done?

- From the linker point of view, a function call like:
  ```
  m.Invoke();
  ```
  is just calling a function called Invoke with the pointer to m as the first parameter.
- The linker is not smart enough to distinguish a class member function and a regular function.
- Therefore, it is supposed to be ok to assign a function pointer to a member function of a sub-class to a function pointer of a member function of the base class.  In the previous example, assigning Application::SetRedBackground to GuiBase::func in the MenuItem class.
- Pre-C++11 language specification was too strict to allow this kind of assignment.

# guibase.h

## Changes

```
#include <functional>

    class MenuItem
    {
    public:
        std::string msg;
        int fskey;
        std::function <void (void)> func;
    };
```

A functional object as a call-back function

```
    template <class T>
    void AddMenu(const std::string &msg,int fskey,void (T::*func)(void),T *owner)
    {
        MenuItem m;
        m.msg=msg;
        m.fskey=fskey;
        m.func=std::bind(func,owner);
        menu.push_back(m);
    }
```

AddMenu is a template function so that it can register a member function of any class.

# guibase.cpp

## Changes

(AddMenu is deleted since it is a template function defined in guibase.h)

```
void GuiBase::OnKeyDown(int fskey)
{
    for(auto m : menu)
    {
        if(m.fskey==fskey)
        {
            m.func();
        }
    }
}
```

# application.h

## Changes

```
class Application : public GuiBase
{
public:
    float bgColor[4];

    virtual void OnInitialize(void);
    virtual void OnDraw(void);
    virtual void OnKeyDown(int fskey);

    void SetRedBackground(void);
    void SetGreenBackground(void);
    void SetBlueBackground(void);
};
```

# application.cpp

## Changes

```
void Application::OnInitialize(void)
{
    bgColor[0]=1;
    bgColor[1]=1;
    bgColor[2]=1;
    bgColor[3]=1;

    AddMenu("R-key  Red"  ,FSKEY_R,&Application::SetRedBackground,this);
    AddMenu("G-key  Green",FSKEY_G,&Application::SetGreenBackground,this);
    AddMenu("B-key  Blue", FSKEY_B,&Application::SetBlueBackground,this);
}


void Application::SetRedBackground(void)
{
    bgColor[0]=1;    bgColor[1]=0;    bgColor[2]=0;
}
void Application::SetGreenBackground(void)
{
    bgColor[0]=0;    bgColor[1]=1;    bgColor[2]=0;
}
void Application::SetBlueBackground(void)
{
    bgColor[0]=0;    bgColor[1]=0;    bgColor[2]=1;
}
```

## Extension

- Imagine you have a mesh viewer.
- You want to make a mesh editor out of the viewer.

- But, you want to keep the viewer as viewer, not an editor.
- You want to have two separate products, one is purely a viewer, and the other is for editing.

- How can you do that?

## Extension

- Two possible options:
  - By inheritance.
  - By making an extension class.
- By inheritance:
  - Create ExtendedApplication class by inheriting Application class.
  - singleton.cpp must be written separately for the original application and extended application.
- By extension class:
  - The Application class must be modified so that it connects with the extension class.
  - The original application has an empty extension class.

## Extension by inheritance

- main.cpp
- guibase.cpp
- guibase.h

Should be a library

- application.cpp
- application.h

Should be another library

- extension.cpp
- extention.h

Written for the extended application

- singleton.cpp

Written separately for the original and extended application

## extension.h

```
#ifndef EXTENSION_IS_INCLUDED
#define EXTENSION_IS_INCLUDED

#include "application.h"

class ExtendedApplication : public Application
{
public:
    virtual void OnInitialize(void);
    void SetWhiteBackground(void);
};

#endif
```

# extension.cpp

```
#include <fslazywindow.h>
#include "extension.h"

void ExtendedApplication::OnInitialize(void)
{
    Application::OnInitialize();
    AddMenu("W-key White",FSKEY_W,&ExtendedApplication::SetWhiteBackground,this);
}

void ExtendedApplication::SetWhiteBackground(void)
{
    bgColor[0]=1;
    bgColor[1]=1;
    bgColor[2]=1;
}
```

First add menu from the original application

Then add its own extension.

Since the call-back function is bound to std::function, it can bind a member function of ExtendedApplication.

## singleton.cpp

Instead of application.h

Changed from Application *

```
#include "extension.h"

static ExtendedApplication *appPtr=nullptr;

GuiBase *GuiBase::GetGuiApplication(void)
{
    if(nullptr==appPtr)
    {
        appPtr=new ExtendedApplication;
    }
    return appPtr;
}
```

Changed from Application

## Extension by an extension class

- main.cpp
- guibase.cpp
- guibase.h

Should be a library

- application.cpp
- application.h
- singleton.cpp

Should be another library

- extension.cpp
- extention.h

Written for the original and extended applications separately

## Extension by an extension class

- In this case, Application and Extension must be mutually connected.
- Menus from the extension must be added after the original Application.

## extension.h

```
#ifndef EXTENSION_IS_INCLUDED
#define EXTENSION_IS_INCLUDED

#include "application.h"

class Extension
{
public:
    Application *appPtr;
    static Extension *GetExtension(void);

    void SetUpMenu(Application &app);
    void SetYellowBackground(void);
};

#endif
```

# extension.cpp

```
#include <fslazywindow.h>

#include "extension.h"

static Extension *extPtr=nullptr;

Extension *Extension::GetExtension(void)
{
    if(nullptr==extPtr)
    {
        extPtr=new Extension;
    }
    return extPtr;
}

void Extension::SetUpMenu(Application &app)
{
    appPtr=&app;
    app.AddMenu("Y-key Yellow",FSKEY_Y,&Extension::SetYellowBackground,this);
}

void Extension::SetYellowBackground(void)
{
    appPtr->bgColor[0]=1;
    appPtr->bgColor[1]=1;
    appPtr->bgColor[2]=0;
}
```

Application class assumes this function is written in one of the CPP files in the project.

## application.h

```
#ifndef APPLICATION_IS_INCLUDED
#define APPLICATION_IS_INCLUDED
/* { */

#include "guibase.h"

class Application : public GuiBase
{
friend class Extension;

public:
    class Extension *extPtr;
    float bgColor[4];

    virtual void OnInitialize(void);
    virtual void OnDraw(void);
    virtual void OnKeyDown(int fskey);

    void SetRedBackground(void);
    void SetGreenBackground(void);
    void SetBlueBackground(void);
};

/* } */
#endif
```

To make a bi-directional connection.

# application.cpp

```
#include <ysclass.h>
#include <ysgl.h>
#include <ysglcpp.h>
#include <ysglslcpp.h>
#include <fslazywindow.h>

#include "application.h"
#include "extension.h"

void Application::OnInitialize(void)
{
    bgColor[0]=1;
    bgColor[1]=1;
    bgColor[2]=1;
    bgColor[3]=1;

    AddMenu("R-key  Red"  ,FSKEY_R,&Application::SetRedBackground,this);
    AddMenu("G-key  Green",FSKEY_G,&Application::SetGreenBackground,this);
    AddMenu("B-key  Blue", FSKEY_B,&Application::SetBlueBackground,this);

    extPtr=Extension::GetExtension();
    extPtr->SetUpMenu(*this);
}
```

(No change in the rest of the file)

Retain a pointer of the Extension, and also call SetUpMenu to allow the extension add menus.

## Canvas, Menu, Dialog, and Widgets

- The examples shown in the previous slides have only textual menu explaining which key does what.
- To make it practical, you need to add more features, such as pull-down menus, dialogs, buttons, list-boxes, etc., called widgets.
- It is not impossible to do.  Each of these is just a combination of simple steps.

## FsGui3D framework

- My solution to the toolkit-rotting problem.
- Keeping it compact so that it goes unmaintainable.
- FsSimpleWindow and FsLazyWindow frameworks as the low-level interface.
- Draws widgets with OpenGL.

- Let’s make a polygonal-mesh editor with this toolkit.