
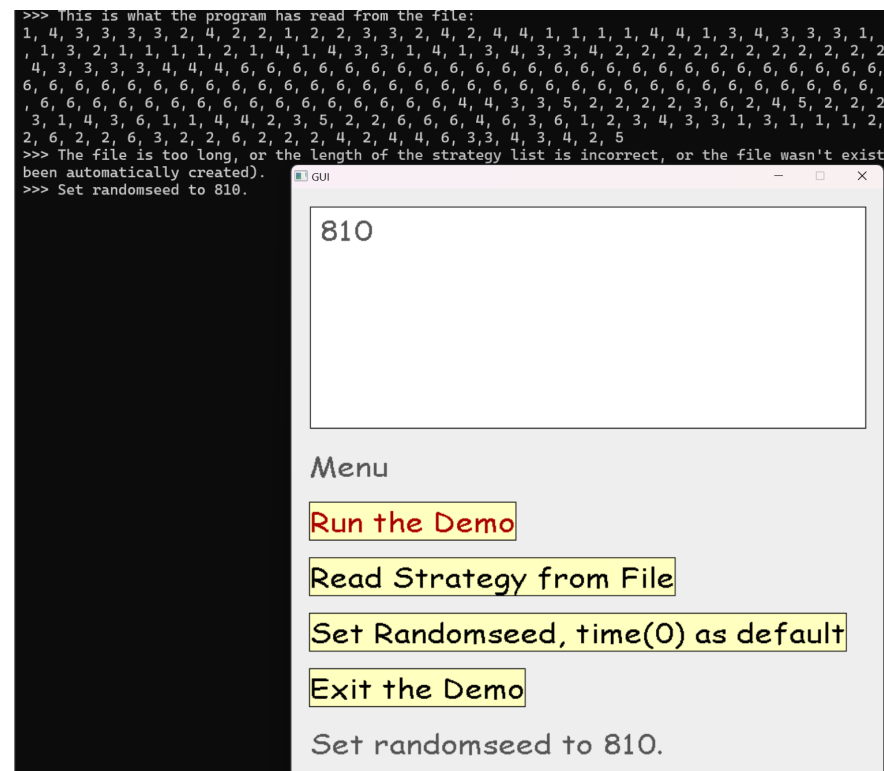


# 程序设计范式和GUI设计简介

复旦大学 2022级 傅全通

# 目录

- 演示GUI程序
- 简介程序设计的各种范式
- 在C语言里实现面向对象编程
- 转向C++实现简单的面向对象编程
- GUI设计的范式  
(以模型——视图——控制器模式为例)
- 用“几乎是C的C++”设计GUI程序



```
class button
{
    // typedef std::function<void(void)> invoker_t;

private:
    int left, top, right, bottom;
    std::string title;
    invoker_t invoker;
    COLORREF textcolor;

public:
    // The most straightforward ctor; but it's diffi
    button(int l, int t, int r, int b, std::string t
    { ...
```

# 面向过程(procedural)

*这个无需多说，我们十分熟悉。*

如果要设计的过程过于复杂，时刻铭记“**自顶向下，逐步求精**”

用 **函数**(function，面向对象编程的基本单元) 隐藏数据和算法的好处：

1. 让以后的工作更加轻松。  
你只需要使用一个之前写的函数就行了，而不是一直记着怎样实现算法逻辑。  
只要你相信该函数对于合法的输入都能正常工作，就可以信任它的输出而不需要记得它是如何工作的。
1. 一旦你能够信任某个函数“可以工作”（比如库函数），就可以开始一遍遍地使用它来写代码解决问题。  
你无需担心任何细节（像如何访问棋盘），这样就可以专注于解决新的问题（比如如何实现AI）。
2. 如果发现逻辑中有个错误，不需要修改代码中的很多地方，只需要修改一个函数而已。
3. 通过函数来隐藏数据结构，你同样也会增强自己存储和表现数据的灵活性。  
你可以先用效率不高但是便于编写的方式，需要时再替换成更快速高效的实现方式；  
完成这些只需要修改少数几个函数，别的都不用动。

# 面向对象(object-oriented)

下学期要学，故这里只是简单介绍其重要概念。

**对象(object，另译作“物件”)**是对现实生活中的物体或事情的**抽象**。

- 对象包含两个含义：**数据**和**动作**。对象则是数据和动作的结合体。
- 对象不仅能够进行**操作**，同时还能够及时**记录下操作结果**。

**继承**和**多态**稍显复杂，暂不涉及；**封装**将在后面以现实例子重点论说。

温馨提示:PI并不严格限制要交纯C语言

不过不要勉强自己学习其他语言，本Pre只用到少量、基本的C++

# 函数式(functional)编程

- 脱胎于 $\lambda$ 运算，将电脑运算视为函数的计算  
如：函数  $f(x) = x + 2$  可以用lambda演算表示为  $\lambda x. x+2$
- 相比于过程化编程，函数式编程里的函数可**就地声明、随时调用**。
- 函数是“一等公民”，可以作为函数的参数或返回值，形成**高阶函数**
- 惰性计算，即表达式不是在绑定到变量时立即计算，而是在求值程序需要产生表达式的值时进行计算

C++中写法: `auto f = [捕获列表] (参数列表) ->返回值类型 {函数体}`

Java中写法: `var f = (参数列表) -> {函数体}`

Python中写法: `f = lambda 参数列表: 函数体`

- 高阶函数：接受函数，乃至返回函数

C语言有接受函数指针作为参数的库函数：

```
void __cdecl qsort(void *_Base, size_t _NumOfElements, size_t _SizeOfElements,  
                  int (__cdecl *_PtFuncCompare)(const void *, const void *));  
//该函数接受一个函数指针，后者接受两个待比较元素，并返回其比较结果
```

Python、C++11 等甚至可以将函数作为函数的返回值

```
def foo(num): # 该函数返回一个函数  
    return lambda x: x*num  
  
print( (foo(10)) (9) ) # 输出90
```

# 泛型（以std::sort为例）

这是一个**函数模板**，接受两个**类型参数**：

随机访问迭代器（指向数组元素的指针）和\_comp仿函数(functor)

```
template<typename _RandomAccessIterator, typename _Compare> inline void
    sort(_RandomAccessIterator __first, _RandomAccessIterator __last,
         _Compare __comp) { }
//__comp仿函数也是一个函数模板哦！*大概* 长这样：
template<typename _Tp>
__comp(const _Tp& lhs, const _Tp& rhs){
    // 可能*lhs和*rhs之间根本不存在小于运算，这时候需要自己写比较函数
    return lhs < rhs;
}
```

熟悉吗？其实就是前面 qsort 和 \_PtfFuncCompare 的C++版本。

# 由泛型产生的模板元编程（元编程的一种）

一个简单的C++模板元编程的例子：

```
template <int N> //模板
struct fib { enum{ x = fib<N - 1>::x + fib<N - 2>::x }; };

template <> //全特化模板
struct fib<1> { enum { x = 1 }; };

template <> //全特化模板
struct fib<0> { enum { x = 0 }; };
```

在程序中的 `fib<20>` 这类表达式会在编译期计算出。



## (例) 如何在C语言里用面向对象设计程序?

(5/2) 想象一个棋盘代表局势和黑白双方的移动（跟PJ2很像！）。  
这个过程其实是**确定棋盘中数据的存储方式**。

```
typedef enum { EMPTY_SQUARE, WHITE_PAWN /* 其他变量 */ } ChessPiece;  
typedef enum PlayerColor { PC_WHITE, PC_BLACK } PlayerColor;  
typedef struct ChessBoard  
{  
    ChessPiece board[ 8 ][ 8 ];  
    PlayerColor whose_move;  
} ChessBoard;
```

提问：为什么在C语言里要用typedef?  
前两个enum的语法有什么区别?

创建**操作棋盘**的函数，都把**指向棋盘的指针**作为第一个参数：

```
ChessPiece getPiece (const ChessBoard *p_board, int x, int y){
    return p_board->board[ x ][ y ];
}
PlayerColor getMove (const ChessBoard *p_board){
    return p_board->whose_move;
}
void makeMove(ChessBoard* p_board, int from_x, int from_y, int to_x, int to_y){
    // 通常情况下, 我们首先需要写点代码验证移动棋子的合法性
    p_board->board[to_x][to_y] = p_board->board[from_x][from_y];
    p_board->board[from_x][from_y] = EMPTY_SQUARE;
}
```

提问：为什么这里第一个参数都是指向棋盘的指针？  
为什么前两个函数的指针加了“**\* 前的const**”？

它们当做其他任何函数一样使用：

```
ChessBoard b; // 首先需要用后面介绍的“构造函数”来恰当地初始化棋盘  
  
getMove( & b );  
makeMove( & b, 0, 0, 1, 0 );
```

事实上，C语言程序员使用这种方式已经很多年了。

然而，这些函数只是与ChessBoard结构体相关联，因为它们恰巧把ChessBoard作为一个参数。

没有地方明确表示：“这个函数应该被当做该结构体的核心部分。”

不过，结构体既包含数据又包含操纵数据的函数，这么做倒挺不错。于是这种语法就被加入了C++。

# 进阶！“真正的”面向对象

把函数变成方法(C++):

```
struct ChessBoard{  
    ChessPiece board[ 8 ][ 8 ];  
    PlayerColor whose_move;  
    ChessPiece getPiece (int x, int y){ return board[ x ][ y ]; }  
    PlayerColor getMove (){ return this->whose_move; }  
    void makeMove (int from_x, int from_y, int to_x, int to_y){  
        board[ to_x ][ to_y ] = board[ from_x ][ from_y ];  
        board[ from_x ][ from_y ] = EMPTY_SQUARE;  
    }  
};
```

不难发现，相比全局函数，方法定义在结构体中，且隐含 `*this` 参数

还可以把结构体中的数据封装，只暴露其方法，这样便符合了(C++中)类的一般形式：

```
class ChessBoard
{
public:
    ChessPiece getPiece(int x, int y);
    PlayerColor getMove();
    void makeMove(int from_x, int from_y, int to_x, int to_y);

private:
    ChessPiece _board[8][8];
    PlayerColor _whose_move;
};
```

方法的定义和之前完全相同；  
只不过要加作用域（类名+双冒号），表示其属于该类。

```
ChessPiece ChessBoard::getPiece(int x, int y)
{
    return _board[x][y];
}
PlayerColor ChessBoard::getMove()
{
    return _whose_move;
}
void ChessBoard::makeMove(int from_x, int from_y, int to_x, int to_y)
{
    // 通常情况下, 首先需要写点代码验证移动棋子的合法性
    _board[to_x][to_y] = _board[from_x][from_y];
    _board[from_x][from_y] = EMPTY_SQUARE;
}
```

# 封装

- 用 `private` 把不想暴露给外界的**数据**和**内部方法**（通常是一些底层的操作）“保护”起来。

（引3）封装 意味着隐藏你的实现（封装它），这样使用类的人只需要处理构成类的接口的那一系列方法就行了。也许使用像“数据隐藏”或者“实现细节”的词组来形容更形象一点，但是“封装”是你时常遇到的术语。

观察上面的例子和阅读这段引文，回答：

- 封装有什么好处？

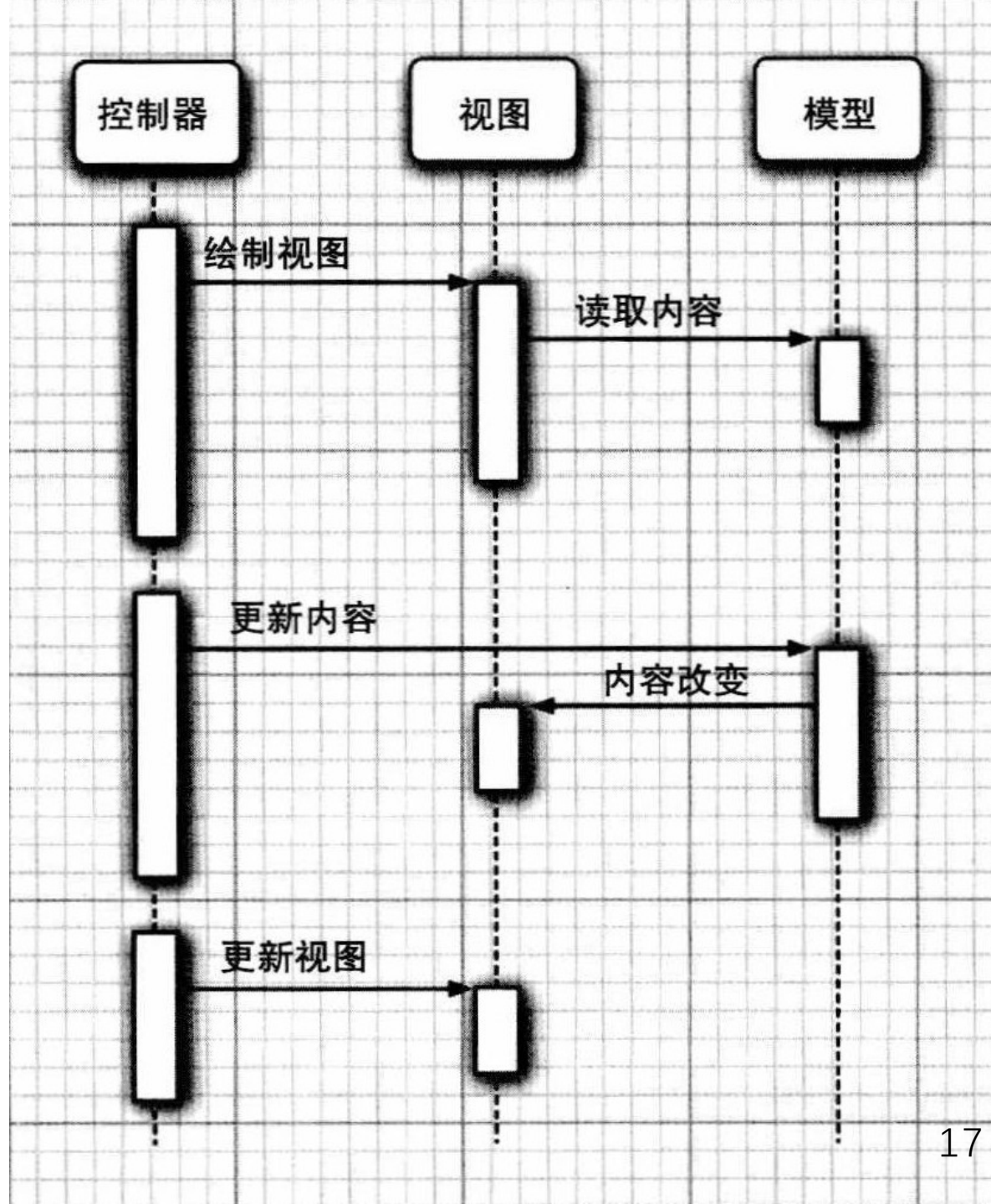
现在你已经大体知道面向对象是什么样子的，  
也看了一个从 `C` 中结构体改进而来的 `C++` 类，  
这为我们进入GUI设计的讨论铺平了道路。



# GUI设计模式

这里介绍Javax.swing提出的  
**模型——视图——控制器**模式  
要求我们提供三个**不同**对象：

- 模型(model): 存储内容。
- 视图(view): 显示内容。
- 控制器(controller):  
处理用户输入，  
如点击鼠标和按下键盘；  
然后决定更改模型或视图。



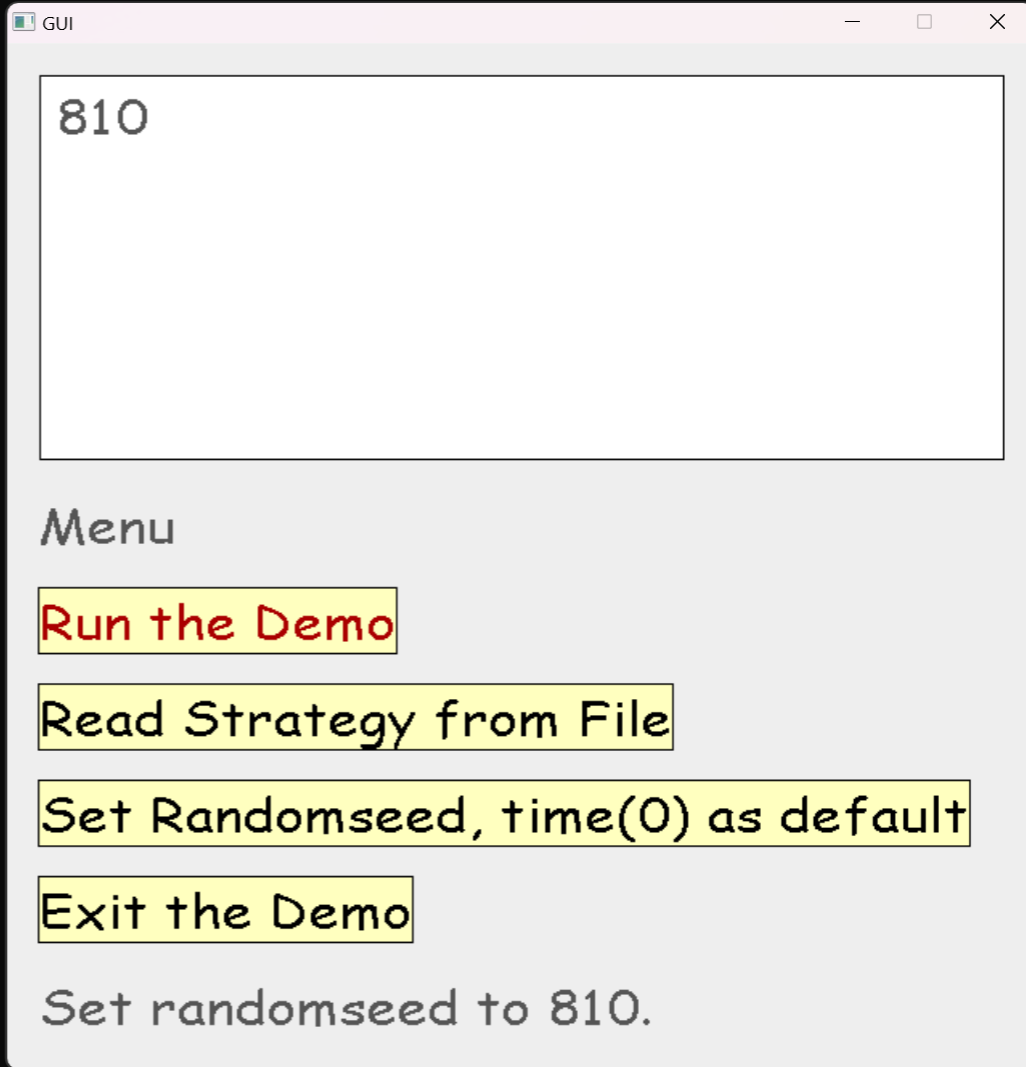
一开始演示的GUI,

视图背后为控制台 (右图),  
其蕴含着模型和控制器。

只需实现**模型**和**控制器**  
(当中的一小部分),  
图形库会帮我们实现**视图**。

下面以按钮类为例。

```
F:\GUI_3.0\GUI.exe X + v
>>> This is what the program has read from the file:
1, 4, 3, 3, 3, 3, 2, 4, 2, 2, 1, 2, 2, 3, 3, 2, 4, 2, 4, 4, 1, 1, 1, 1, 4, 4, 1, 3, 4, 3, 3, 3, 1,
, 1, 3, 2, 1, 1, 1, 1, 2, 1, 4, 1, 4, 3, 3, 1, 4, 1, 3, 4, 3, 3, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
4, 3, 3, 3, 3, 4, 4, 4, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 3, 3, 5, 2, 2, 2, 2, 3, 6, 2, 4, 5, 2, 2, 2,
3, 1, 4, 3, 6, 1, 1, 4, 4, 2, 3, 5, 2, 2, 6, 6, 6, 4, 6, 3, 6, 1, 2, 3, 4, 3, 3, 1, 3, 1, 1, 1, 2,
2, 6, 2, 2, 6, 3, 2, 2, 6, 2, 2, 2, 4, 2, 4, 4, 6, 3, 3, 4, 3, 4, 2, 5
>>> The file is too long, or the length of the strategy list is incorrect, or the file wasn't exist
been automatically created).
>>> Set randomseed to 810.
```



# 实现**button**的模型，即**button**类的**数据部分**

```
class button
{
    // 指向void(void)类型函数的指针
    typedef void (*invoker_t)(void);

    // 按钮的存储数据如下
private:
    int left, top, right, bottom;        // 存储按钮的位置和大小
    std::string /*是字符串*/ title;      // 存储按钮上的文本
    invoker_t invoker;                   // 按钮被点击时的响应动作
    COLORREF /*本质是int*/ textcolor;    // 存下按钮文本的颜色
}
```

提问：问什么需要这个难懂的函数指针？

（其实可以用多态，多态的本质正是每个对象有不同的函数指针）

# button的控制器

```
public:
    // 构造函数 (构造器方法) , 可以向其传入参数以创建指定的button实例
    button(int left, int top, char *txt, COLORREF txtcol, invoker_t ivk_func);

    ~button() { /*析构函数, 用于妥善销毁对象。这里用默认或直接不写即可*/ }

    // 检查鼠标是否在按钮框内部
    bool check(int x, int y){
        return left<=x && x<=right && top<=y && y<=bottom;
    }

    // 在GUI里绘制按钮的样貌
    void show();

    // 调用响应函数, 即处理点击事件
    void on_message() { invoker(); }
};
```

类是抽象的，调用构造函数才会创建类的**实例**，即通常说的“对象”。

可以把这些按钮对象（都是**模型**）放进数组，便于统一操作。

```
button buttons[] = {
    button(LEFT, BODY_1, "Run the Demo",
        [&main_screen](void) -> void // 匿名函数
        {
            puts(">>> Clicked \"Run the Demo\".");
            move();           // 切换到演示界面, 完全复用PJ1
            main_screen();    // 重新绘制主界面
        },
        RED),
    button(LEFT, BODY_2, "Read Strategy" /*, 某个函数*/),
    button(LEFT, BODY_3, "Set Randomseed" /*, 某个函数*/),
    button(LEFT, BODY_4, "Exit", []{ closegraph(); exit(0); }));
```

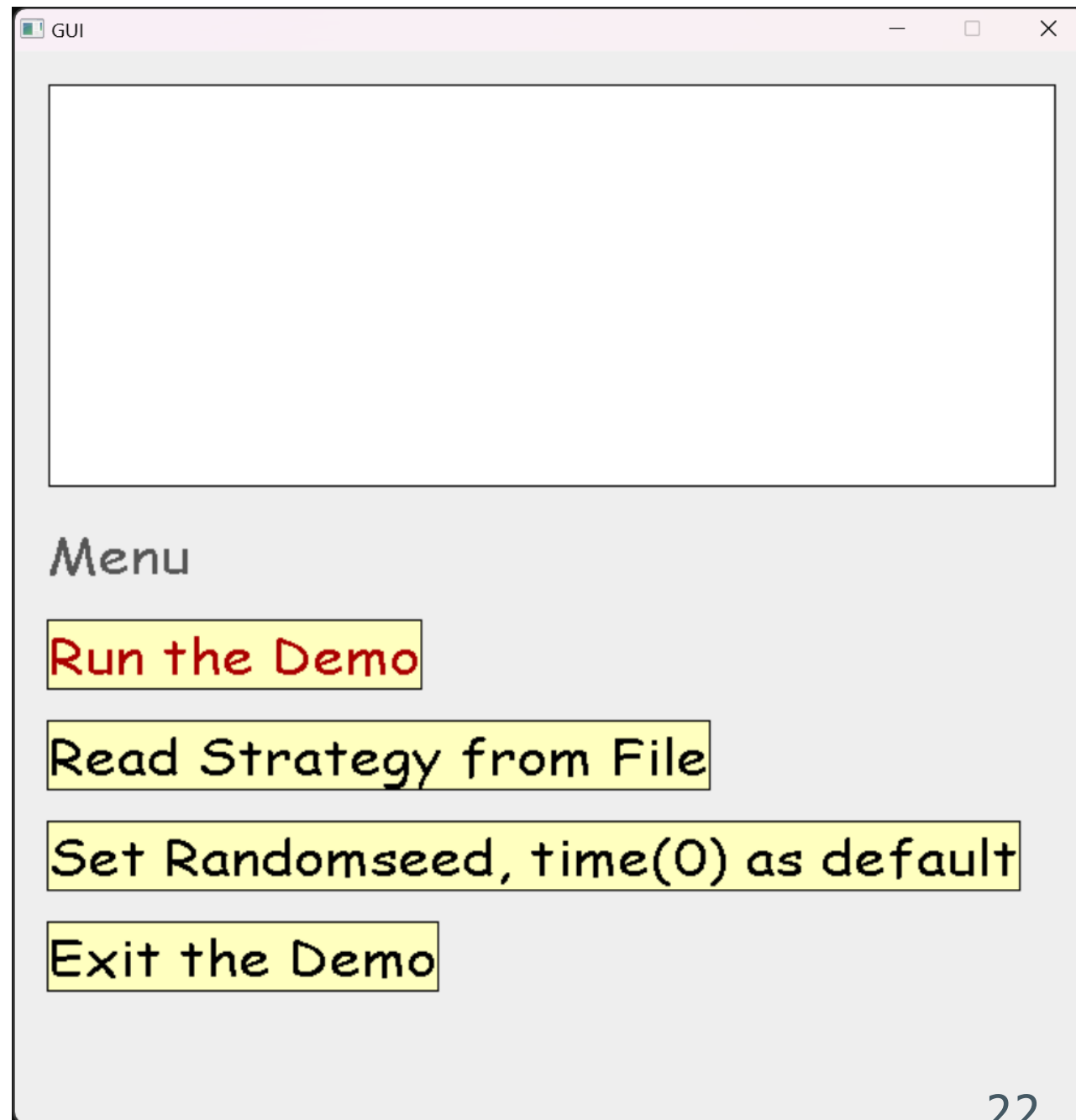
现在可以把它们展现在视图中

```
for (size_t i = 0;  
     i < sizeof(buttons)/sizeof(button);  
     ++i)  
    buttons[i].show();
```

*//C++中简写如下。不深究右值引用*

```
for (auto &&i : buttons) i.show();
```

类似地可以实现文本框类。不再赘述。



## 外层控制器的一般模式：

```
for (ExMessage msg;;)           // 声明消息变量
{
    msg = getmessage(EM_MOUSE);   // 获取消息输入
    if (msg.lbutton)              // 如果左键按下
    {
        for (auto &&i : buttons)   // 遍历每个按钮
            if (i.check(msg.x, msg.y)) // 光标在按钮框内
                i.on_message();    // 按钮按下事件
        // ...其他控件
    }
}
```

至此，GUI设计简介完成

(如果有时间)

介绍EasyX库——Windows下使用C语法的免费(但不开源)简单图形库  
(至少比SDL简单, 而且有详细的中文文档和丰富的实例程序)

旨在帮大家完成PJ中的GUI加分项, 因此偏实用。

`Qt`、`Javax.swing` 或者 `JavaFX` 都是更好的选择,  
但是需要涉及继承、接口等等复杂的面向对象特性,  
所以这里介绍面向过程的EasyX。



# EasyX获取和安装(略说)

## [链接](#)

有任意版本Visual Studio的可以直接打开安装包；其他方式：  
[在 CLion、Dev-C++ 或 Code::Blocks 下面配置 EasyX](#)

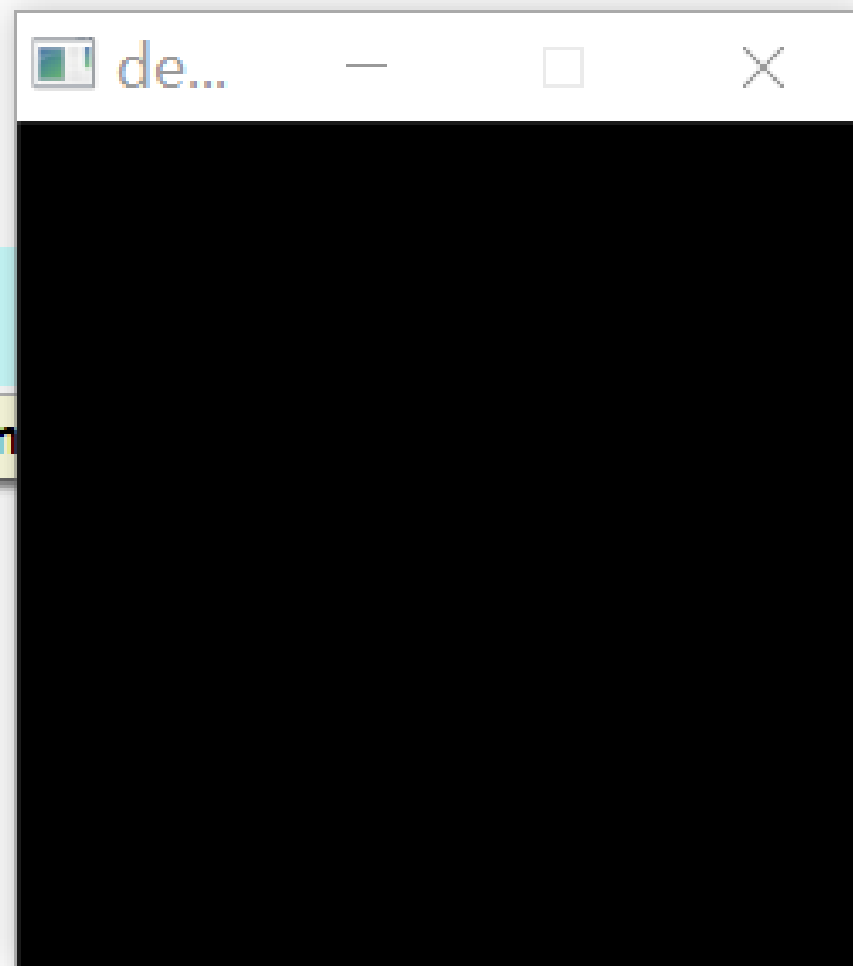
---

EasyX的核心是两个头文件、一个静态库：

`graphics.h`（继承自Turbo C）、`EasyX.h`，`libEasyX.a`

## 创建空白窗口

```
int main(){  
    initgraph(200,200);  
    HWND initgraph (int width, int height, in  
    system("pause");  
}
```



# 常用绘图函数

demo总共只用了这些函数

用法可以望文生义，  
或参考[EasyX在线文档](#)

GUI\_3.0 > demo.cpp > main()

```
1  #include <graphics.h>
2  int __cdecl _system(const char *_Command);
3  int main()
4  {
5      initgraph(600, 400); int left, top, right, bottom;
6
7      setbkcolor(YELLOW); // 背景色
8      cleardevice();
9
10     setlinecolor(BLACK); // 划线和边框的颜色
11     rectangle(left = 250, top = 250, right = 349, bottom = 349);
12
13     setfillcolor(LIGHTGRAY); // 填充的颜色
14     fillrectangle(left = 20, top = 20, right = 199, bottom = 199);
15
16     settextcolor(RED); // 文本的颜色
17     setfont(40, 16, "Microsoft YaHei UI"); // 字体: 微软雅黑
18     /*等同于*/ settextstyle(40, 16, "Microsoft YaHei UI");
19     outtextxy(20, 300, "Not Hello_world"); // 指定位置输出文本
20
21     setfillcolor(GREEN); // 每次fill之前要重新填色
22     solidellipse(left = 300, top = 20, right = 399, bottom = 119);
23
24
25
26
```

只需要把PJ1当中的空格全都换成rectangle  
罐子换成方形、小黄换成椭圆即可

如果你想，可以绘制一个可爱的小黄形象，然后加载进去

*// NULL 表示加载到绘图窗口。*

这样就可以画各种想要的GUI。



```
loadimage(NULL, "Yellow.jpeg");
```

# 消息处理

getMessage() 用于获取一个消息。如果消息队列中没有，就一直等待。

```
ExMessage getMessage(BYTE filter = -1);
```

filter 指定要获取的消息范围，默认 -1 获取所有类别的消息（why? ）  
可以用以下值或值的组合获取指定类别的消息：

标志	描述
EX_MOUSE	鼠标消息。
EX_KEY	按键消息。
EX_CHAR	字符消息。
EX_WINDOW	窗口消息。

```
bool peekmessage(ExMessage *msg, BYTE filter = -1, bool removemsg = true);
```

这个函数用于获取一个消息，并立即返回。

**removemsg** 指定处理完消息后是否从消息队列中移除。设为默认即可。

有关具体各种消息的宏，见[链接](#)。

有了这些函数，**模型、视图、控制器**都可以的用C语言来实现。  
这样就可以动手设计简单的GUI程序。