基础实验

实验1

使用new申请空间

new 语句比 **malloc** 语句对程序员更加友好。 写法上，更加简洁， 也更加好懂。

以矩阵运算为例。 当我们需要创建一个矩阵的时候， 可以使用二级指针， 在内存中申请一块空间。

```
  mat = new int*[row_len] ;

  for(int i=0 ; i<row_len ; i++){

    mat[i] = new int [col_len] ;

  }
```

观察上面的代码：

首先给二级指针**mat**申请内存。 申请了长度为**row_len**的内存，类型是一级**int**指针。

然后给每个一级指针申请动态内存，长度为**col_len**, 类型为**int**。

可以将 **[]** 中的内容看做长度， 之前的 **int***, **int** 等看做类型， 这样申请空间的语句就变得好懂。 多维数组申请空间需要从高级向低级申请。

传递引用

```
int **&mat
```

的写法很让人在意。 从字面上看（从右向左看），这应该是“引用的二级指针”。 但是一般的解释说这是二级指针的引用。

从意义上来说，引用只是一个别名，创建指向引用的指针是不合理的。 这点容易想明白。 但是写法上却有些让人迷惑。

相对的，这样的写法就是错误的：

```
int &**mat
```

实验2.1

实验中创建了两个版本：在函数中传递形参， 在函数中传递引用。 发现传递引用的版本减少了很多对象的构造和析构， 最终保留了传递引用的版本。

对于不带引用的版本， 在报告中保留源代码和输出信息（包含对于构造对象和析构对象的分析）。

源代码

```
#include<iostream>
#include<cstdio>
#include<cmath>
```

```
4
5  class Point
6  {
7  private:
8      /* data */
9      double x, y ;
10 public:
11     Point(double x=0, double y=0)
12     : x(x), y(y)
13     {
14         std::cout<<"Creating a Point...\n" ;
15     }
16     Point (const Point& obj){
17         *this = obj ;
18         std::cout<<"Copying a Point...\n" ;
19     }
20     ~Point(){
21         std::cout<<"Deleting a Point...\n" ;
22     }
23
24     double get_x() const {return x;}
25     double get_y() const {return y;}
26 };
27 double dist(Point A, Point B) {
28         // 返回A和B的euclid距离
29         double dist_x = A.get_x() - B.get_x() ;
30         double dist_y = A.get_y() - B.get_y() ;
31         return sqrt(dist_x*dist_x + dist_y*dist_y) ;
32 }
33
34 class Circle
35 {
36 private:
37     /* data */
38     Point c ;
39     double r ;
40 public:
41     Circle(Point c, double r=0)
42     : c(c), r(r)
43     {
44         std::cout<<"Creating a Circle...\n" ;
45     }
46     Circle(const Circle& obj){
47         *this = obj ;
48         std::cout<<"Copying a Circle...\n" ;
49     }
50     ~Circle(){
51         std::cout<<"Deleting a Circle...\n" ;
52     }
53     Point get_c() const {return c ; }
54     double get_r() const {return r; }
55 };
56
57 bool is_intersect(Circle O1,Circle O2){
58     std::cout<<"\n" ;
59     Point c1, c2 ;
60     c1 = O1.get_c() ;
61     c2 = O2.get_c() ;
```

```
    return O1.get_r() +O2.get_r() > dist(c1, c2) ;
}

int main(){
    Point A(1,0), B(0,1) ;
    std::cout<<"\n" ;
    std::cout<<dist(A,B)<<"\n";
    std::cout<<"\n" ;
    Circle C1(Point(1,0), 2), C2(Point(4,0), 1.2);
    std::cout<<"\n" ;
    std::cout<<is_intersect(C1, C2)<<"\n" ;
    std::cout<<"\n" ;
    C1.get_c() ;
    std::cout<<"\n" ;
    return 0 ;
}

```

输出信息

```
创建A和B

Creating a Point...

Creating a Point...

dist创建形参并且销毁

Copying a Point...

Copying a Point...

1.41421

Deleting a Point...

Deleting a Point...

创建Circle的时候，首先创建临时的Point，然后通过copy复制创建Circle的c，创建Circle，最后删掉临时的Point

Creating a Point...

Copying a Point...

Creating a Circle...

Deleting a Point...

Creating a Point...

Copying a Point...

Creating a Circle...

Deleting a Point...

```

```
is_intersect先创建形参Circle的c，然后通过copy复制创建形参

Creating a Point...

Copying a Circle...

Creating a Point...

Copying a Circle...

创建c1,c2，使用函数get_c 会copy一个实例出来，然后返回，赋值之后删掉。然后给dist传参，这时候要创建形参，用完删掉。删掉c1,c2. 最后删掉O1, O2(当然删掉Circle之后要把c也删掉).

Creating a Point...

Creating a Point...

Copying a Point...

Deleting a Point...

Copying a Point...

Deleting a Point...

Copying a Point...

Copying a Point...

Deleting a Point...

Deleting a Point...

Deleting a Point...

Deleting a Point...

1

Deleting a Circle...

Deleting a Point...

Deleting a Circle...

Deleting a Point...

可以看到，调用get_c也会使用复制构造函数

Copying a Point...

Deleting a Point...

最后删掉main中创建的实例(按照创建时间倒序删除， 使用堆栈来保存创建的对象)

Deleting a Circle...

Deleting a Point...
```

```
94
95 Deleting a Circle...
96
97 Deleting a Point...
98
99 Deleting a Point...
100
101 Deleting a Point...
```

实验2.2

使用new申请内存

```
1  void fun()
2
3  {
4
5    A *a = new A();
6
7  }
8
9  int main()
10
11 {
12
13   while(1)
14
15   {
16
17     fun();
18
19   }
20
21 return 0;
22
23 }
```

当离开fun时，虽然离开了作用域，但用new动态开辟空间的对象是不会析构的，你可以观察任务管理器，看到内存一直在上升。但你在其他地方却无法使用**a**所开辟的空间，因为**a**这个指针是保存在栈上的，当离开作用域后就自动析构(或者说自动消失了)，但它所在分配空间是分配在堆上的，只有主动析构或程序结束，才会释放空间，也就是丢失了这块空间的地址，无法操作这块空间了。

本程序析构函数中，一定要释放矩阵的内存。 不然就会只将指针指向的地址释放掉， 但是堆中相应的空间还是被占用。

Delete 和 delete [] 的区别

https://blog.csdn.net/cbNotes/article/details/38900799

delete 的参数只有一个。 之前使用了这样的语句：

```
1    delete pA1, pA2, pA3 ;
```

结果从输出信息中发现只调用了一个析构函数。 **pA2**, **pA3**其实都没有被**delete**掉。

实验3

Virtual function

当基类中有一种功能，需要在所有的子类中实现， 但是不同的子类实现方法又很不相同，这时候可以使用**virtual function** 的方式声明该函数， 表示这个函数是有的， 但是我并没有明确的定义出来， 你作为子类应该自己想办法定义这个函数。 严格来说， 上面说的是**pure virtual function**, 是父类没有对函数进行定义。

普通的**virtual function** , 父类是可以进行定义的， 但是子类可以根据具体情况进行更改。

当然一般的函数就是**non-virtual function** , 子类是不可以更改的。

```
virtual double getArea() const=0 ; // pure virtual function. No definition
```

Shape 类里面就使用**virtual function** 定义了**getArea**函数， 因为只知道这是一个形状， 对于获取面积不能提供任何信息， 也就没什么可写的， 全部留给子类去完成。

```
virtual double getArea() const

  {

    return a * b;

  }
```

而在**Rectangle**里面， 就可以对**getArea**进行定义了。

后面在**Square**里面， 就没有再对**getArea**进行定义， 而是直接调用了Rectangle里面定义好的函数。

构造函数的调用顺序

Creating a Shape...

Creating a Rectangle...

Creating a Square...

9

Creating a Shape...

Creating a Rectangle...

12

Creating a Shape...

Creating a Circle...

12.5664

Deleting a Circle...

Deleting a Shape...

Deleting a Rectangle...

Deleting a Shape...

Deleting a Square...

Deleting a Rectangle...

Deleting a Shape...

这里获取了产生各个对象并且输出面积的时候， 构造函数的调用顺序。 可以看到， 构造函数是由内而外进行调用的。 父类先于子类。 因为子类包含了父类的一部分， 要先构造内部， 才能在外面进行进一步的构造。

实验5.1

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 **= 0** 来指定的。

设计**抽象类**（通常称为 **ABC**）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为**接口**使用。如果试图实例化一个抽象类的对象，会导致编译错误。

抽象类中会定义好一个能够对外界开放的函数， 并且将其定义为虚函数， 要由子类实现具体的内容。 这样规范了统一的接口形式， 而且便于灵活开发。

之前在实验3中，已经使用了**pure virtual function** 去定义**Shape**中的**getArea**.

```
1   Shape *S2 = &r1;
2
3   std::cout << S2->getArea() << "\n\n";
```

这里使用一个**Shape**类型的指针访问一个**Rectangle**类型的对象。 我们想要使用**Rectangle**类型的**getArea**计算面积并且返回值。在没有加上**virtual**的时候， 由于指针是Shape**类型**，程序调用了**Shape**类型的**getArea**。

如果使用了**virtual**关键字，程序将根据引用或指针指向的 **对象类型** 来选择方法，否则使用**引用类型或指针类型**来选择方法。

使用了**virtual** 之后， 程序就能够正常调用**Rectangle**中重新定义的**getArea**了

综合实验

设计说明

数据库中用户信息格式

name(key, unique)	password	pet_num	pet_list	total_fight	win_fight
QString	QString	int	QString	int	int

server端发送消息类型和格式

list[0]	list[1]	list[2]	list[3]	list[4]	list[5]
z	true/false				
d	true/reload/false	pet_num	pet_list	total_fight	win_fight
a	all_user_list				
o	online_user_list				
detail	pet_num	pet_list	total_fight	win_fight	
rand_p	random_pet_list				

client端发送消息类型和格式

list[0]	list[1]	list[2]	list[3]	list[4]
z	name	password	pet_list	
d	name	password		
a				
o				
detail	name			
rand_p				
result	pet_num	pet_list	total_fight	win_fight

详细说明

pet_list

字符串类型。每个精灵的信息最后，用 | 分隔。

每个精灵的信息包括 编号，名字，等级，经验。 信息之间用 * 分隔。

user_list

字符串类型。包含一串用户的name. 信息之间用 * 分隔。

random_pet_list

字符串类型。包含一串随机精灵的编号。信息之间用 * 分隔。

Server端架构

程序开始运行后，实例化TcpServer类(继承自QMainWindow)的对象win. 也就是Server的主界面。

在win的构造函数中，实例化一个server_m类(继承自QTcpServer)的对象server.

窗口只有一个按钮"开启服务器"， 按下之后，server开始监听。

在server的构造函数中，启动数据库User(如果不存在则创建)，并试图在数据库中创建table user(如果已经存在则不执行操作)。

如果有client试图建立连接，则会触发server的incomingConnection函数，分配一个socket描述符descriptor，server会实例化一个新的socketthread类(继承自QThread)的对象thread, 然后thread开始运行。

在thread的构造函数中，实例化一个socket_m类(继承自QTcpSocket)的对象socket, 其socket描述符即为传递过来的descriptor； 同时该线程中的对象sql打开数据库User. 多个线程共享同一个数据库。

每个thread有一个socket。每当有一个用户登陆的时候，server就会多打开一个thread。程序使用多线程异步通信方式。

之后每个线程进入等待消息的状态。如果接收到来自client的消息，则进行处理。共有7种消息：注册，登录，请求所有用户的列表，请求在线用户列表，请求某用户的详细数据，让服务器发送15个随机的精灵序号，更新数据库。

Client 端架构

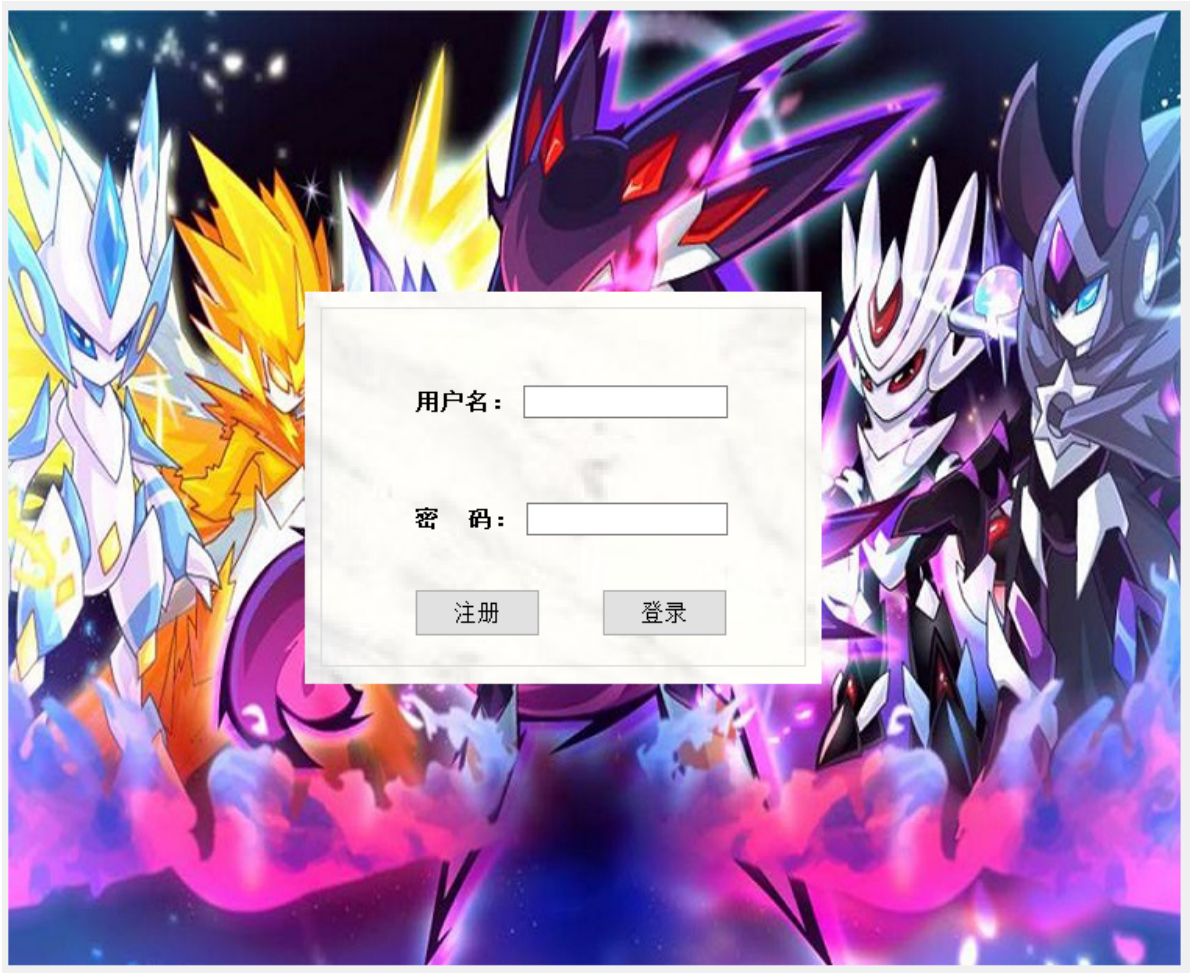
Client端共有两个窗体类： TcpClient 和 FightManager (都是继承自QMainWindow) . 而TcpClient是Client端的主窗口。

TcpClient窗体

TcpClient窗体共有四个页面：

1. 登录/注册界面
2. 查看信息界面
3. 比赛准备界面
4. 送出精灵界面

1.登录/注册界面



用户选择登录或者注册。如果注册已经存在的用户名，则会提示用户名已被注册。登录时，如果用户名不存在，或者密码错误，则会提示用户名或密码错误。如果尝试在多客户端上登录同一个账号，则会提示该用户已登录。

用户第一次注册账号时，系统会提示注册成功。这时点击登录，则可以进入游戏。登陆成功之后，socket会试图和server端建立连接。连接使用固定的ip(127.0.0.1)和port(8000) . 注册，登陆不成功，或者主动退出账号，server端都会断开连接。

2.查看信息界面

在这里，用户可以查看自己的用户名，比赛胜率和获得的荣誉。

左上方的表格显示的是用户自己的精灵列表。单击表格中的某一行可以在右侧表格中显示该精灵的详细信息。

左下方的表格显示其他用户的详细信息。用户可以通过上方的两个下拉列表选择。左边的下拉列表可以选择请求全部用户的名单或者在线用户的名单； 右边的下拉列表可以选择 用户名单中的某一位用户，显示他的详细信息。

点击刷新，则可以刷新当前页面上的信息(可能有新用户登录游戏，在线用户列表变化；精灵升级赛获胜，获得经验，等等)。

点击升级对战和决斗，都会进入页面3.

升级对战获胜后可以获得经验，失败则无事发生。

决斗赛胜利后可以获得对战的敌方精灵(1级)，失败后则挑选用户列表中的三只精灵，由用户选择一只送出(若是不够3只则全部列出)。

点击退出返回到界面1，关闭正在通信的socket.

3. 比赛准备界面

选定自己的某个精灵，并选择升级对战或者决斗之后，会进入比赛准备界面。左边的列表会显示服务器随机发送过来的15个精灵，等级从1到15级递增。单击某一行，可以在右边的列表中显示该精灵的详细信息。

单击刷新，服务器会送来新的15个精灵的列表，并显示在屏幕上。

单击开始对战，将会跳转到FightManager窗口，进行战斗演示和结算。

单击返回，则返回到界面2.

4.送出精灵界面

在决斗赛中失败之后，会跳转到送出精灵界面。列表中显示系统从用户的精灵列表中随机抽取处的三个精灵(如果不足三个则全部抽取)的名称和等级。 单击某一行可以在左边的方框中显示该精灵的图片。单击确认送出，则会将目前选择的精灵从用户的精灵列表中移除，并回到界面2。

如果用户送出精灵之后，目前已经没有精灵，系统则会随机分发给用户一个初始状态的精灵(1级)。

FightManager窗体

两只精灵会按照自身的敏捷度确定攻击间隔，然后自动进行攻击。有可能两只精灵同时发动攻击，系统会在下方的信息栏中进行提示。

下方的列表显示了战斗的详细信息。基本格式为

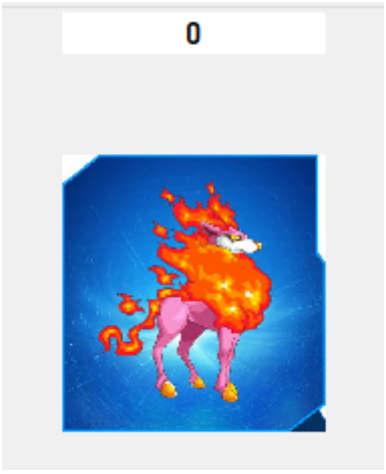
pet_name 发动攻击 skill_name 攻击类型 是否暴击

pet_name 受到攻击 skill_name 伤害值 是否属性克制 是否闪避

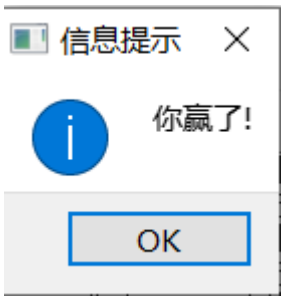
当精灵受到伤害的时候，上方的精灵图片会发生抖动，血量条会减少。

其中一只精灵死亡后，战斗结束

```
谱尼 发动攻击 落芳天华          闪避增益   暴击:YES
谱尼 受到攻击 normalAttack      伤害:50    属性克制:NO          躲避:NO
里奥斯 受到攻击 落芳天华        伤害:40    属性克制:NO          躲避:NO
里奥斯 死亡
```



战斗胜利或者失败之后，会弹出相应的提示框。



如果战斗失败，则会返回到主窗口的页面3； 如果战斗胜利，则会留在战斗窗口，玩家可以单击重新开始，开始一局新的战斗。

问题与解决

1

编写自定义类，使用自定义slot函数的时候，系统报错。如果要定义自己的槽函数，必须要定义类的时候，加上Q_OBJECT宏定义，问题不再出现。

```
class XXX
{
    Q_OBJECT
```

2

server的online_user是被所有thread共享的。线程在访问online_user的时候，必须要实现线程同步。

- 当有thread正在读取online_user的时候，不能有thread对online_user进行写操作。
- 当有thread正在写入online_user的时候，不能有其他thread写入online_user, 同时不能有thread读取online_user.

程序中使用QSemaphore 实现此功能。

```
int readcount=0;
QSemaphore mutex(1),wrt(1);
```

```
wrt.acquire();  // 同一时间只能有一个thread修改user (server_m 中的QStringList online_user , 被所有的thread共享)
user->append(myname);
wrt.release();
```

wrt的权限仅有一个。

当有一个thread使用了wrt的权限之后，其他想写入online_user的thread就只能挂起，等待wrt的权限被释放。保证了同一时间只有一个thread写入online_user.

```
//读取在线用户列表
        mutex.acquire();            //互斥访问
        readcount++;
        if(readcount == 1)
            wrt.acquire();
        mutex.release();

        QStringList ulist=*user;    //读操作
        sendData+="#";
        for(int i=0;i<ulist.size();i++)
        {
            sendData+=ulist[i];
            sendData+="*";
        }
        mutex.acquire();            //读者减一
        readcount--;
        if(readcount==0)            //若此为最后一个读者，则释放写信号量
            wrt.release();
        mutex.release();
```

readcount 记录了正在读取online_user的thread的个数。

mutex的作用不是保证同一时间只有一个thread读取online_user, 而是保证第一个读取online_user的用户能够拿到wrt权限，同时最后一个reader释放掉wrt的权限。也就是，当有reader的时候，不能写入online_user ; 所有的reader都退出的时候，可以有thread写入online_user.

同时，如果此时有writer正在写入online_user, 那么想要读取online_user的thread就会在 wrt.acquire() 处被挂起，直到wrt的权限被释放。保证了当有thread正在写入online_user的时候不能有thread读取online_user.

注： 数据库User也是被所有的thread以及server共享的，但是在需要更新User的时候，没有使用互斥锁，因为每个thread都代表着不同的用户，操作的肯定是数据库User中的不同部分，不存在资源访问冲突的情况。

3

阻塞模式下，在I/O操作完成前，执行的操作函数一直等候而不会立即返回，该函数所在的线程会阻塞在这里。相反，在非阻塞模式下，套接字函数会立即返回，而不管I/O是否完成，该函数所在的线程会继续运行。

程序中实现的是多线程并行运行，每个线程单socket通信，所以适合阻塞式socket通信。而Qt中的QTcpSocket实现的是非阻塞式socket通信。为此，使用waitForBytesWritten函数实现阻塞式socket通信。该函数会等待socket将信息完全写入缓冲区之后，再向下继续运行。函数的默认等待时间是30s, 也可以手动设置。

测试程序如下

```
QTcpSocket tmpSock;
char* buf ="hello";
tcpSock.connectToHost("59.64.159.87",7716);
tcpSock.write(buf,strlen(buf)+1);
Sleep(3000);
tcpSock.disconnect();
```

在执行write函数之后，进程立即进行休眠，导致线程未能向缓冲区写入数据。

在write函数之后加入waitForBytesWritten函数，线程会向缓冲区写完数据之后再进行休眠。

4

在应用中插入图片的方式：

```
    QPixmap pix_big(":/pictures/img/bg.jpeg");
    int w = ui->label_pic_big->width();
    int h = ui->label_pic_big->height();
    ui->label_pic_big->setPixmap(pix_big.scaled(w,h));
    QPixmap pix_small(":/pictures/img/white.jpeg");
    w = ui->label_pic_big->width();
    h = ui->label_pic_big->height();
    ui->label_pic_small->setPixmap(pix_small.scaled(w,h));
```

界面上设置固定大小的QLabel, 然后向QLabel中插入图片。

5

Qt中的QString类型字符串提供了非常方便的字符串操作方式。

QString类型可以和QString类型的字符串或者const char * 拼接。

可以使用.split() 方法将字符串按照关键字进行分割，返回QStringList. split中可以设置分割的关键字，程序中使用 | 和 * 作为分割的关键字。

附录A:基础部分程序源代码

实验1

```
#include <cstdio>
#include <iostream>

// 矩阵的行数和列数
const int row_len = 4;
const int col_len = 5;

void get_memory(int **&mat){    // 申请内存空间
    mat = new int*[row_len] ;
    for(int i=0 ; i<row_len ; i++){
        mat[i] = new int [col_len] ;
```

```
    }
}

void release_memory(int **&mat){    // 释放内存空间
    for(int i=0 ; i<row_len ; i++){
        delete mat[i] ;
    }
    delete mat ;
}

void read_in_mat(int **&mat)
{

    for (int i = 0; i < row_len; i++)
    {
        for (int j = 0; j < col_len; j++)
        {
            std::cin >> mat[i][j];
        }
    }
    std::cout<<"\n" ;
    return;
}

void print_mat(int **&mat)
{
    for (int i = 0; i < row_len; i++)
    {
        for (int j = 0; j < col_len; j++)
        {
            std::cout << mat[i][j] << " ";
        }
        std::cout << "\n";
    }
    std::cout<<"\n" ;
    return;
}

void mat_add(int **&mat1, int **&mat2, int **&mat3)
{
    for (int i = 0; i < row_len; i++)
    {
        for (int j = 0; j < col_len; j++)
        {
            mat1[i][j] = mat2[i][j] + mat3[i][j];
        }
    }
    return;
}

void mat_sub(int **&mat1, int **&mat2, int **&mat3)
{
    for (int i = 0; i < row_len; i++)
    {
        for (int j = 0; j < col_len; j++)
        {
            mat1[i][j] = mat2[i][j] - mat3[i][j];
        }
```

```
    }
    return;
}



int main()
{
    int **A1, **A2, **A3 ;
    get_memory(A1) ;
    get_memory(A2) ;
    get_memory(A3) ;

    read_in_mat(A1) ;
    read_in_mat(A2) ;

    mat_add(A3, A1, A2) ;
    print_mat(A3) ;

    mat_sub(A3, A1, A2) ;
    print_mat(A3) ;

    release_memory(A1) ;
    release_memory(A2) ;
    release_memory(A3) ;

    print_mat(A1) ;
    return 0;
}
```

实验2.1

```
#include<iostream>
#include<cstdio>
#include<cmath>

class Point
{
private:
    /* data */
    double x, y ;

public:
    Point(const double& x=0, const double& y=0)
    : x(x), y(y)
    {
        std::cout<<"Creating a Point...\n" ;
    }
    Point (const Point& obj){
        *this = obj ;
        std::cout<<"Copying a Point...\n" ;
    }
    ~Point(){
        std::cout<<"Deleting a Point...\n" ;
    }

```

```cpp
    double get_x() const {return x;}
    double get_y() const {return y;}

    Point& operator = (const Point& p) {
        x = p.x ;
        y = p.x ;
        return *this ;
    }
};

double dist(const Point& A,const Point& B) {
        // 返回A和B的euclid距离
        double dist_x = A.get_x() - B.get_x() ;
        double dist_y = A.get_y() - B.get_y() ;
        return sqrt(dist_x*dist_x + dist_y*dist_y) ;
}

class Circle
{
private:
    /* data */
    Point c ;
    double r ;
public:
    Circle(const Point& c, const double& r=0)
    : c(c), r(r)
    {
        std::cout<<"Creating a Circle...\n" ;
    }
    Circle(const Circle& obj){
        *this = obj ;
        std::cout<<"Copying a Circle...\n" ;
    }
    ~Circle(){
        std::cout<<"Deleting a Circle...\n" ;
    }
    Point get_c() const {return c ; }
    double get_r() const {return r; }
};

bool is_intersect(const Circle& O1, const Circle& O2){
    std::cout<<"\n" ;
    return O1.get_r() +O2.get_r() > dist(O1.get_c(), O2.get_c()) ;
}

int main(){
    Point A(1,0), B(0,1) ;
    std::cout<<"\n" ;
    std::cout<<dist(A,B)<<"\n";
    std::cout<<"\n" ;
    Circle C1(Point(1,0), 2), C2(Point(4,0), 1.2);
    std::cout<<"\n" ;
    std::cout<<is_intersect(C1, C2)<<"\n" ;
    std::cout<<"\n" ;
    C1.get_c() ;
    std::cout<<"\n" ;
    return 0 ;
}
```

实验2.2

```
#include<iostream>
#include<cstdio>
#include<iomanip>

class Mat
{
private:
    /* data */
    int lines, rows ;
    int **mat ;
public:
    Mat(const int& lines, const int& rows)
    : lines(lines), rows(rows)
    {
        mat = new int* [lines] ;
        for(int i=0 ; i<lines ; i++){
            mat[i] = new int [rows] ;
        }
        std::cout<<"Creating a Mat. Apply for memory...\n" ;
    }
    Mat (const Mat& obj){
        lines = obj.lines ;
        rows = obj.rows ;
        mat = new int* [lines] ;
        for(int i=0 ; i<lines ; i++){
            mat[i] = new int [rows] ;
        }
        for(int i=0 ; i<lines ; i++ ){
            for(int j=0 ; j<rows ; j++ )
            mat[i][j] = obj.mat[i][j] ;
        }
        std::cout<<"Copying a Mat. Apply for memory...\n" ;
    }
    ~Mat() {
        for(int i=0 ; i<lines ; i++){
            delete mat[i] ;
        }
        delete mat ;
        std::cout<<"Deleting a Mat. Release memory...\n" ;
    }

    int get_lines() const {return lines; }
    int get_rows() const {return rows; }

    void read_in(){
        for(int i=0 ; i<lines ; i++ ){
            for(int j=0 ; j<rows ; j++ )
            std::cin>>mat[i][j] ;
        }
        std::cout<<"\n";
    }

    void print_out() const {
```

```
        for(int i=0 ; i<lines ; i++ ){
            for(int j=0 ; j<rows ; j++ ){
                std::cout<<std::setw(10)<<mat[i][j]<<" " ;
            }
            std::cout<<"\n" ;
        }
        std::cout<<"\n" ;
    }

    Mat
    operator + (const Mat& A) const {
        Mat re(lines, rows) ;
        for(int i=0 ; i<lines ; i++) {
            for(int j=0 ; j<rows ; j++) {
                re.mat[i][j] = mat[i][j] + A.mat[i][j] ;
            }
        }
        return re ;
    }

    Mat
    operator - (const Mat& A) const {
        Mat re(lines, rows) ;
        for(int i=0 ; i<lines ; i++) {
            for(int j=0 ; j<rows ; j++) {
                re.mat[i][j] = mat[i][j] - A.mat[i][j] ;
            }
        }
        return re ;
    }

    Mat&
    operator = (const Mat& A){
        for(int i=0 ; i<lines ; i++) {
            for(int j=0 ; j<rows ; j++) {
                mat[i][j] = A.mat[i][j] ;
            }
        }
    }
};

const int lines=3, rows=3 ;

int main(){

    Mat A1(lines,rows), A2(lines,rows), A3(lines,rows) ;
    A1.read_in() ;
    A2.read_in() ;

    A3 = A1 + A2 ;
    A3.print_out() ;
    A3 = A1 - A2 ;
    A3.print_out() ;

    Mat *pA1, *pA2, *pA3 ;
    pA1 = new Mat(lines,rows) ;
    pA2 = new Mat(lines,rows) ;
    pA3 = new Mat(lines,rows) ;
```

```
    pA1->read_in() ;
    pA2->read_in() ;

    *pA3 = *pA1 + *pA2 ;
    pA3->print_out() ;
    *pA3 = *pA1 - *pA2 ;
    pA3->print_out() ;

    delete pA1 ;
    delete pA2 ;
    delete pA3 ;
    return 0 ;
}
```

实验3

```
#include <iostream>
#include <cmath>

class Shape
{
public:
    Shape()
    {
        std::cout << "Creating a Shape...\n";
    }
    Shape(const Shape &obj)
    {
        *this = obj;
        std::cout << "Copying a Shape...\n";
    }
    ~Shape()
    {
        std::cout << "Deleting a Shape...\n";
    }
    virtual double getArea() const
    {
        std::cout << "Calling shape getarea...\n";
        return 0 ;
    }
};

class Rectangle
    : public Shape
{
protected: // 可以被子类继承
    double a, b;

public:
    Rectangle(const double &a, const double &b)
        : a(a), b(b)
    {
        std::cout << "Creating a Rectangle...\n";
    }
    Rectangle(const Rectangle &obj)
```

```
    {
        *this = obj;
        std::cout << "Copying a Rectangle...\n";
    }
    ~Rectangle()
    {
        std::cout << "Deleting a Rectangle...\n";
    }

    virtual double getArea() const
    {
        std::cout << "Calling rectangle getarea...\n";
        return a * b;
    }
};

const double Pi = 3.1415926;

class Circle
    : public Shape
{
private:
    double r;

public:
    Circle(const double &r)
        : r(r)
    {
        std::cout << "Creating a Circle...\n";
    }
    Circle(const Circle &obj)
    {
        *this = obj;
        std::cout << "Copying a Circle...\n";
    }
    ~Circle()
    {
        std::cout << "Deleting a Circle...\n";
    }

    virtual double getArea() const
    {
        std::cout << "Calling circle getarea...\n";
        return Pi * r * r;
    }
};

class Square
    : public Rectangle
{
private:
    double x;

public:
    Square(const double &x)
        : Rectangle(x, x), x(x)
    {
        std::cout << "Creating a Square...\n";
```

```
    }
    Square(const Square &obj)
        : Rectangle(x, x)
    {
        *this = obj;
        std::cout << "Copying a Sqaure...\n";
    }
    ~Square()
    {
        std::cout << "Deleting a Square...\n";
    }
};

int main()
{
    Square s1(3);
    std::cout << s1.getArea() << "\n\n";
    Rectangle r1(3, 4);
    std::cout << r1.getArea() << "\n\n";
    Circle c1(2);
    std::cout << c1.getArea() << "\n\n";
    Shape *S2 = &r1;
    std::cout << S2->getArea() << "\n\n";
    return 0;
}

```

实验4

```
#include<iostream>
#include<cstdlib>
#include<ctime>
int main(){
    srand((unsigned) time(0)) ;
    int price = 1+rand()%1000 ;
    std::string input ;
    int price_input ;
    int guess_cnt=10 ;
    // 总共10次有效的猜数的机会
    // 一个理智的player 应该能在10次的时候猜出答案
    while(guess_cnt--){
        std::cout<<"Please input an integer within [1,1000]: \n" ;
        std::cin>>input ;
        price_input = atoi(input.c_str()) ;
        if( ! (price_input>=1 && price_input<=1000)){
            std::cout<<"Wrong input!!\n" ;
            guess_cnt++ ;
        }
        else {
            if(price_input > price){
                std::cout<<"Too big.\n" ;
            }
            else if (price_input < price){
                std::cout<<"Too small.\n" ;
            }
            else {
```

```
                std::cout<<"Bingo! You get the right answer.\n" ;
                return 0 ;
            }
        }
    }
    std::cout<<"You've used up all your chances!\n" ;
    return 0 ;
}
```

实验5.1

```
#include <iostream>
#include <cmath>

class Shape
{
public:
    Shape()
    {
        std::cout << "Creating a Shape...\n";
    }
    Shape(const Shape &obj)
    {
        *this = obj;
        std::cout << "Copying a Shape...\n";
    }
    ~Shape()
    {
        std::cout << "Deleting a Shape...\n";
    }
    virtual double getArea() const =0 ;
    // virtual double getArea() const
    // {
    //  std::cout << "Calling shape getarea...\n";
    //  return 0 ;
    // }
};

class Rectangle
    : public Shape
{
protected: // 可以被子类继承
    double a, b;

public:
    Rectangle(const double &a, const double &b)
        : a(a), b(b)
    {
        std::cout << "Creating a Rectangle...\n";
    }
    Rectangle(const Rectangle &obj)
    {
        *this = obj;
        std::cout << "Copying a Rectangle...\n";
    }
    ~Rectangle()
```

```
    {
        std::cout << "Deleting a Rectangle...\n";
    }

    virtual double getArea() const
    {
        std::cout << "Calling rectangle getarea...\n";
        return a * b;
    }
};

const double Pi = 3.1415926;

class Circle
    : public Shape
{
private:
    double r;

public:
    Circle(const double &r)
        : r(r)
    {
        std::cout << "Creating a Circle...\n";
    }
    Circle(const Circle &obj)
    {
        *this = obj;
        std::cout << "Copying a Circle...\n";
    }
    ~Circle()
    {
        std::cout << "Deleting a Circle...\n";
    }

    virtual double getArea() const
    {
        std::cout << "Calling circle getarea...\n";
        return Pi * r * r;
    }
};

class Square
    : public Rectangle
{
private:
    double x;

public:
    Square(const double &x)
        : Rectangle(x, x), x(x)
    {
        std::cout << "Creating a Square...\n";
    }
    Square(const Square &obj)
        : Rectangle(x, x)
    {
        *this = obj;
```

```
        std::cout << "Copying a Sqaure...\n";
    }
    ~Square()
    {
        std::cout << "Deleting a Square...\n";
    }
};

int main()
{
    Square s1(3);
    std::cout << s1.getArea() << "\n\n";
    Rectangle r1(3, 4);
    std::cout << r1.getArea() << "\n\n";
    Circle c1(2);
    std::cout << c1.getArea() << "\n\n";
    Shape *S2 = &r1;
    std::cout << S2->getArea() << "\n\n";
    return 0;
}
```

实验5.2

```
#include<iostream>

class Point
{
private:
    /* data */
    double x, y ;
public:
    Point(const double& x, const double& y )
    : x(x), y(y)
    {
        std::cout<<"Creating a Point...\n" ;
    }
    Point(const Point& obj){
        *this = obj ;
        std::cout<<"Copying a Point...\n" ;
    }
    ~Point()
    {
        std::cout<<"Deleting a Point...\n" ;
    }
    void print_Point() {
        std::cout<<"("<<x<<","<<y<<")\n" ;
    }
    Point& operator ++ () {
        x+=1 ;
        y+=1 ;
        return *this ;
    }

    Point& operator -- () {
```

```
        x-=1 ;
        y-=1 ;
        return *this ;
    }
    // 重载后置++, -- 需要额外的int占位参数
    // 为了绕过语法限制
    Point operator ++ (int) {
        Point re = *this ;
        x+=1 ;
        y+=1 ;
        return re ;
    }

    Point operator -- (int) {
        Point re = *this ;
        x-=1 ;
        y-=1 ;
        return re ;
    }
};

int main(){
    Point p(0,0) ;
    // Point *ptr ;
    // ptr = &p ;
    // std::cout<<ptr<<std::endl ;
    // ptr = &(++p) ;
    // std::cout<<ptr<<std::endl ;
    // ptr = &(p++) ;
    // std::cout<<ptr<<std::endl ;

    (p++).print_Point() ;
    (p--).print_Point() ;
    (++p).print_Point() ;
    (--p).print_Point() ;

    return 0 ;
}
```