从模块到面向对象

问题

用"*"在屏幕上显示一个形状如$y = a\sin(x)+c$的图形。

或者，用"*"在屏幕上显示一个形状如y = x%100的图形。

问题不在于图形函数表达，而是，程序设计求解时的思维方式。

1. 朴素的想法

y=sin(x)，定义域和值域不合适用来绘制，所以需要进行平移和缩放，形状为：

y = coeff_y * (sin(2.0 * c_pi / num_x_per_cycle * x) + 1.0)

简单起见，把图形旋转90°，这样每一行只有一个*

或者周期函数

y = x % num_per_cycle;

G0-普通的解

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

const double c_pi = acos(-1.0);

void Draw0(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G0-------------------------" << std::endl;
    int rows = num_x_per_cycle * cycles;
    for (auto y = 0; y < rows; y++)
    {
        int x;
        {
#ifdef _Easy_mode
            x = y % num_x_per_cycle;
#else
            double theta = 2.0 * c_pi / num_x_per_cycle * y;
            x = int(coeff_y * (sin(theta) + 1.0) + 0.5);//+0.5是为了四舍五入
#endif
        }

        for (auto i = 0; i <= x; i++) std::cout << " ";
        std::cout << "*" << std::endl;
    }
    std::cout << "-------------------------G0" << std::endl;
}
```

首先封装数学技巧。把sin(x)缩放封装成MyFun函数，从而简化Draw1中的逻辑。

G1-封装sin函数

```
//图形栅格化
double MyFun(double x, double num_x_per_cycle, double coeff_y)
{
#ifdef _Easy_mode
    return  int(x) % int(num_x_per_cycle);
#else
    return coeff_y * (sin(2.0 * c_pi / num_x_per_cycle * x) + 1.0);
#endif
}

void Draw1(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G1-------------------------" << std::endl;

    int rows = num_x_per_cycle * cycles;
    for (auto row = 0; row < rows; row++)
    {
        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
        for (auto i = 0; i <= x; i++) std::cout << " ";
        std::cout << "*" << std::endl;
    }

    std::cout << "-------------------------G1" << std::endl;
}
```

现在存在一个明显的问题，必须严格按照'*'在屏幕位置一行一行输出，从左到右，从上到下，像打字机一样的工作方式。

如果同时需要画多个图形，需要计算好从左往右空格和*的关系，非常麻烦。

可以采用一种缓存的机制：1首先先将内容存到缓存中。2最后将缓存内容显示出来。

步骤1中，因为绘制实际上是对内存的操作，和绘制的位置没有关系。首先引入一个string的单行缓存。

G2-单行缓存机制

```
void Draw2(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G2-------------------------" << std::endl;

    int rows = num_x_per_cycle * cycles;
    int width = coeff_y * 2 + 4;
    for (auto row = 0; row < rows; row++)
    {
        //创建buffer，buffer清空成背景' '
        std::string buffer(width, ' ');

        //字符写到buffer
        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
        buffer[x] = '*';

        //buffer输出到屏幕
        std::cout << buffer << std::endl;
    }

```

```
    std::cout << "-------------------------G2" << std::endl;
}
```

G2存在很大限制，需要一行一行进行显示（即不能先第2行画一个字符，然后再回到第1行画另一个字符）。

于是，可以引入一个自然的想法：多行缓存机制。

采用动态内存机制来实现多个std::string对象：std::string *buffer = new std::string[rows+1];申请内存，用完之后通过delete[] buffer;释放

G3-多行缓存机制-动态数组

```
void Draw3(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G3-------------------------" << std::endl;

    int rows = num_x_per_cycle * cycles;
    std::string *buffer = new std::string[rows];

    int width = coeff_y * 2 + 4;
    //buffer初始化，背景清空成' '
    for (auto row = 0; row < rows; row++)
    {
        buffer[row].resize(width, ' ');
    }

    //字符写入buffer
    for (auto row = 0; row < rows; row++)
    {
        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
        buffer[row][x] = '*';
    }

    //buffer输出到屏幕
    for (auto row = 0; row < rows; row++)
    {
        std::cout << buffer[row] << std::endl;
    }

    delete[] buffer;
    std::cout << "-------------------------G3" << std::endl;
}
```

动态内存的管理显得有些麻烦，可以采用C++中的vector容器

G3-多行缓存机制-vector

```
void Draw3_vector(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G3-vector-------------------------" << std::endl;

    int rows = num_x_per_cycle * cycles;
    std::vector<std::string> buffer;
    buffer.resize(rows);

```

```
    int width = coeff_y * 2 + 4;
    //buffer初始化，背景清空成' '
    for (auto row = 0; row < rows; row++)
    {
        buffer[row].resize(width, ' ');
    }

    //字符写入buffer
    for (auto row = 0; row < rows; row++)
    {
        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
        buffer[row][x] = '*';
    }

    //buffer输出到屏幕
    for (auto row = 0; row <= rows; row++)
    {
        std::cout << buffer[row] << std::endl;
    }

    std::cout << "-------------------------G3-vector" << std::endl;
}
```

此时，可以比较便利地画图，甚至画多个sin(x)

G4-多图

```
//同时画多个图形
void Draw4(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G4-------------------------" << std::endl;

    int rows = num_x_per_cycle * cycles;
    std::vector<std::string> buffer;
    buffer.resize(rows);

    int width = coeff_y * 2 + 4;
    //buffer初始化，背景清空成' '
    for (auto row = 0; row < rows; row++)
    {
        buffer[row].resize(width, ' ');
    }

    //字符写入buffer
    for (auto row = 0; row < rows; row++)
    {
        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
        buffer[row][x] = '*';
    }
    for (auto row = 0; row < rows; row++)
    {
        int x = int(MyFun(row, num_x_per_cycle, coeff_y*0.5) + 0.5);
        buffer[row][x] = '*';
    }

    //buffer输出到屏幕
    for (auto row = 0; row < rows; row++)
```

```
    {
        std::cout << buffer[row] << std::endl;
    }

    std::cout << "--------------------------G4" << std::endl;
}
```

2. 模块封装

更进一步，我们可以把这个缓存机制封装出来

封装出一个存放数据的数据结构struct RenderBuffer，以及相关的操作。

G5-封装

```
struct RenderBuffer{
    int                             width;
    int                             height;
    std::vector<std::string>        buffer;
};

struct RenderBuffer* InitBuffer(int width, int height)
{
    RenderBuffer* rbuffer = new RenderBuffer;
    rbuffer->width = width;
    rbuffer->height = height;
    rbuffer->buffer.resize(height);

    for (auto y = 0; y < height; y++)
    {
        rbuffer->buffer[y].resize(width, ' ');
    }
    return rbuffer;
}

void ShowBuffer(struct RenderBuffer* rbuffer)
{
    for (auto y = 0; y < rbuffer->height; y++)
    {
        std::cout << rbuffer->buffer[y] << std::endl;
    }
}

void WriteBuffer(struct RenderBuffer* rbuffer, double x0, double y0, char c)
{
    int x = int(x0 + 0.5);
    int y = int(y0 + 0.5);

    if (x < 0 || x >= rbuffer->width || y < 0 || y >= rbuffer->height)
        return; //越界

    rbuffer->buffer[y][x] = c;
}
```

调用这些模块时大概如下所示。定义功能和使用功能被隔离开来。并且WriteBuffer1函数中的判断，可以防止字符位置不在许可范围导致的错误。

```
void Draw5(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G5----------------" << std::endl;

    //注意：控制台窗口宽度有限制，会自动换行，所以设置小一些
    //创建，并初始化
    struct RenderBuffer* rbuffer = InitBuffer(100, 65);

    //图形写到buffer
    for (auto y = 0; y <= num_x_per_cycle * cycles; y++)
    {
        double x = MyFun(y, num_x_per_cycle, coeff_y);
        WriteBuffer(rbuffer, x, y, '*');
    }

    //buffer显示到屏幕
    ShowBuffer(rbuffer);

    delete rbuffer;

    std::cout << "----------------G5" << std::endl;
}
```

模块分离之后的好处是，可以升级某个模块，而不影响其它模块，比如，我们把buffer机制更新成效率更高的单个std::string

此外，我们不希望使用动态内存。因此可以引入“&”引用。

G6-封装-单std::string

```
struct RenderBuffer_single_string {
    int             width;
    int             height;
    std::string     buffer;
};

void InitBuffer(int width, int height, struct RenderBuffer_single_string&
rbuffer)
{
    rbuffer.width = width;
    rbuffer.height = height;
    rbuffer.buffer.resize(height * (width + 1), ' ');
    for (auto y = 0; y < height; y++)
    {//每隔width个字符设置一个换行符'\n'
        rbuffer.buffer[y * (width + 1) + width] = '\n';
    }
}

void ShowBuffer(const struct RenderBuffer_single_string& rbuffer)
{
    std::cout << rbuffer.buffer << std::endl;
}

void WriteBuffer(struct RenderBuffer_single_string& rbuffer, double x0,
double y0, char c)
{
```

```
    int x = int(x0 + 0.5);
    int y = int(y0 + 0.5);
    if (x < 0 || x >= rbuffer.width || y < 0 || y >= rbuffer.height)
        return; //越界
    rbuffer.buffer[y * (rbuffer.width + 1) + x] = c;
}
```

调用时

```
void Draw6(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G6----------------" << std::endl;

    //创建buffer
    struct RenderBuffer_single_string rbuffer;
    //初始化buffer
    InitBuffer(100, 65, rbuffer);

    //字符写到buffer
    for (auto y = 0; y <= num_x_per_cycle * cycles; y++)
    {
        double x = MyFun(y, num_x_per_cycle, coeff_y);
        WriteBuffer(rbuffer, x, y, '*');
    }

    //buffer显示到屏幕
    ShowBuffer(rbuffer);

    std::cout << "----------------G6" << std::endl;
}
```

其中，使用了一个技巧，通过在一个std::string中间添加'\n'，以达到显示多行字符的效果。

采用引用之后，内存不需要额外管理(std::string不需要额外管理)，非常简洁。以下代码绘制了2个图形，一个是竖着显示的sin，另一个是横着显示的sin

G7-多图-横向和竖向sin

```
void Draw7(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G7-------------------------" << std::endl;
    //创建buffer
    struct RenderBuffer_single_string rbuffer;
    //初始化buffer
    InitBuffer(100, 65, rbuffer);

    //竖着的sin
    for (auto y = 0; y <= num_x_per_cycle * 0.8 * cycles; y++)
    {
        double x = MyFun(y, num_x_per_cycle * 0.8, coeff_y);
        WriteBuffer(rbuffer, x, y, '*');
    }
    //横着的sin
    for (auto x = 0; x <= num_x_per_cycle * cycles; x++)
    {
```

```
        double y = MyFun(x, num_x_per_cycle, coeff_y * 0.8);
        WriteBuffer(rbuffer, x, y, '*');
    }

    //buffer显示到屏幕
    ShowBuffer(rbuffer);

    std::cout << "-------------------------G7" << std::endl;
}
```

3. 类的封装

数据结构和操作数据结构的过程封装在一起，那就是类了！

G8-类的封装

```
class CRenderBuffer
{
private:
    int             width;
    int             height;
    std::string     buffer;
public:
    void Init(int width0, int height0)
    {
        width = width0;
        height = height0;
        buffer.resize(height * (width + 1), ' ');
        for (auto y = 0; y < height; y++)
        {//每隔width个字符设置一个换行符'\n'
            buffer[y * (width + 1) + width] = '\n';
        }
    }

    void Show()
    {
        std::cout << buffer << std::endl;
    }

    void Write(double x0, double y0, char c)
    {
        int x = int(x0 + 0.5);
        int y = int(y0 + 0.5);
        if (x < 0 || x >= width || y < 0 || y >= height)
            return; //越界，不画
        buffer[y * (width + 1) + x] = c;
    }
};
```

类是一种抽象数据类型，包括数据以及操作这些数据的过程。类定义出来的变量称为对象。面向对象程序设计，主要功能都是通过对象实现。

对象中包含了具体的数据，也可以发起对函数的调用。

```
void Draw8(int num_x_per_cycle, int cycles, int coeff_y)
{
    std::cout << "G8------------------------------" << std::endl;
    //创建CRenderBuffer的对象rbuffer
    CRenderBuffer rbuffer;
    //初始化buffer，通过对象rbuffer发起方法调用，初始化buffer
    rbuffer.Init(100, 65);

    for (auto y = 0; y <= num_x_per_cycle * cycles; y++)
    {
        double x = MyFun(y, num_x_per_cycle, coeff_y);
        rbuffer.Write(x, y, '*');
    }

    rbuffer.Show();
    std::cout << "------------------------------G8" << std::endl;
}
```

4. 面向对象技术

G9-面向对象

首先需要要层级结构，把可能变化的功能/函数抽象成接口，定义成多态函数。以此应对变化。此设计实际上是一种面向未来变化的机制。在这个机制之下，每个未来的修改，都可以定位到基类某个特定的接口之中；不变的部分则信息隐藏。

```
class IRenderBuffer
{
protected:
    int             width;
    int             height;
public:
    virtual void Init(int width0, int height0)=0;
    virtual void Show() = 0;
    virtual void Write(double x0, double y0) = 0;
};
```

然后派生出具体的实现子类，比如之前例子所用的方法

```
class CRenderBuffer : public IRenderBuffer
{
private:
    std::string     buffer;
public:
    void Init(int width0, int height0)
    {
        width = width0;
        height = height0;
        buffer.resize(height * (width + 1), ' ');
        for (auto y = 0; y < height; y++)
        {//每隔width个字符设置一个换行符'\n'
            buffer[y * (width + 1) + width] = '\n';
        }
    }

    void Show()
```

```
    {
        std::cout << buffer << std::endl;
    }

    void Write(double x0, double y0)
    {
        int x = int(x0 + 0.5);
        int y = int(y0 + 0.5);
        if (x < 0 || x >= width || y < 0 || y >= height)
            return; //越界，不画
        buffer[y * (width + 1) + x] = '*';
    }
};
```

应用时，buffer不需要指定具体类型，在运行时，它会自动判定自己属于哪个类型，并调用相应类的方法

```
void Draw9(IRenderBuffer &buffer, int num_x_per_cycle, int cycles, int
coeff_y)
{
    std::cout << "G9----------------------------------------" << std::endl;

    //初始化buffer，通过对象rbuffer发起方法调用，初始化buffer
    buffer.Init(100, 65);

    for (auto y = 0; y < num_x_per_cycle * cycles; y++)
    {
        double x = MyFun(y, num_x_per_cycle, coeff_y);
        buffer.Write(x, y);
    }
    buffer.Show();
    std::cout << "----------------------------------------G9" << std::endl;
}
```

有了这种层次结构，可以派生新的类，而不需要改变Draw_sin9函数。CRenderBuffer_image讲绘制结果存储到一个图像result.jpg中

```
class MRGB
{
public:
    int R;
    int G;
    int B;
public:
    MRGB()
    {
        R = G = B = 0;
    }
    MRGB(int r, int g, int b)
    {
        R = r, G = g, B = b;
    }
};

class CRenderBuffer_image : public IRenderBuffer
{
```

```
private:
    std::vector<struct MRGB>    buffer;
public:
    void Init(int width0, int height0)
    {
        width = width0;
        height = height0;
        buffer.resize(width * height);
    }

    void Show()
    {
        FIBITMAP* bitmap = FreeImage_Allocate(width, height, 24);
        if (!bitmap)
        {
            return;
        }

        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                MRGB mrgb = buffer[y * width + x];
                RGBQUAD rgb;
                rgb.rgbRed = mrgb.R, rgb.rgbGreen = mrgb.G, rgb.rgbBlue =
mrgb.B;
                FreeImage_SetPixelColor(bitmap, x, y, &rgb);//设置像素值
            }
        }

        //存图像
        std::string file = "result.jpg";
        FreeImage_Save(FIF_PNG, bitmap, file.c_str());
        FreeImage_Unload(bitmap);//释放FreeImage
    }

    void Write(double x0, double y0)
    {
        int x = int(x0 + 0.5);
        int y = int(y0 + 0.5);
        if (x < 0 || x >= width || y < 0 || y >= height)
            return; //越界，不画

        buffer[y * width + x] = MRGB(0, 255, 0);
    }
};

```

main函数调用时，对Draw_sin9调用接口一摸一样，只是传递的IRenderBuffer子类对象不同。

```
int main()
{
    int num_x_per_cycle = 30;
    int cycle = 2;
    int coeff_y = 20;

    {//过去的方法
```

```
        CRenderBuffer rbuffer;
        Draw9(rbuffer, num_x_per_cycle, cycle, coeff_y);
    }
    {//一个新的变化，由于采用了面向对象技术，无需修改使用对象的函数Draw_sin9()，轻松接入系统
        CRenderBuffer_image imageBuffer;
        Draw9(imageBuffer, num_x_per_cycle, cycle, coeff_y);
    }
}
```