

# 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的用来绘制，所以需要进行平移和缩放，形状为：

$y = \text{coeff\_y} * (\sin(2.0 * \text{c\_pi} / \text{num\_x\_per\_cycle} * x) + 1.0)$

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

$y = x \% \text{num\_per\_cycle};$

## GO-普通的解

```
1 #include <iostream>
2 #include <cmath>
3
4 const double c_pi = acos(-1.0);
5
6 void Draw0(int num_x_per_cycle, int cycles, int coeff_y)
7 {
8     std::cout << "G0-----" << std::endl;
9     int rows = num_x_per_cycle * cycles;
10    for (auto y = 0; y < rows; y++)
11    {
12        int x;
13        {
14            #ifdef _Easy_mode
15                x = y % num_x_per_cycle;
16            #else
17                double theta = 2.0 * c_pi / num_x_per_cycle * y;
18                x = int(coeff_y * (sin(theta) + 1.0) + 0.5); //+0.5是为了四舍五入
19            #endif
20        }
21
22        for (auto i = 0; i <= x; i++) std::cout << " ";
23        std::cout << "*" << std::endl;
24    }
25    std::cout << "-----G0" << std::endl;
26 }
```

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

```
1 //图形栅格化
2 double MyFun(double x, double num_x_per_cycle, double coeff_y)
3 {
4 #ifdef _Easy_mode
5     return int(x) % int(num_x_per_cycle);
6 #else
7     return coeff_y * (sin(2.0 * c_pi / num_x_per_cycle * x) + 1.0);
8 #endif
9 }
10
11 void Draw1(int num_x_per_cycle, int cycles, int coeff_y)
12 {
13     std::cout << "G1-----" << std::endl;
14
15     int rows = num_x_per_cycle * cycles;
16     for (auto row = 0; row < rows; row++)
17     {
18         int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
19         for (auto i = 0; i <= x; i++) std::cout << " ";
20         std::cout << "*" << std::endl;
21     }
22
23     std::cout << "-----G1" << std::endl;
24 }
```

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

```
1 void Draw2(int num_x_per_cycle, int cycles, int coeff_y)
2 {
3     std::cout << "G2-----" << std::endl;
4
5     int rows = num_x_per_cycle * cycles;
6     int width = coeff_y * 2 + 4;
7     for (auto row = 0; row < rows; row++)
8     {
9         //创建buffer, buffer清空成背景' '
10        std::string buffer(width, ' ');
11
12        //字符写到buffer
13        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
14        buffer[x] = '*';
15
16        //buffer输出到屏幕
17        std::cout << buffer << std::endl;
18    }
19 }
```

```

20     std::cout << "-----G2" << std::endl;
21 }

```

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

```

1 void Draw3(int num_x_per_cycle, int cycles, int coeff_y)
2 {
3     std::cout << "G3-----" << std::endl;
4
5     int rows = num_x_per_cycle * cycles;
6     std::string *buffer = new std::string[rows];
7
8     int width = coeff_y * 2 + 4;
9     //buffer初始化，背景清空成' '
10    for (auto row = 0; row < rows; row++)
11    {
12        buffer[row].resize(width, ' ');
13    }
14
15    //字符写入buffer
16    for (auto row = 0; row < rows; row++)
17    {
18        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
19        buffer[row][x] = '*';
20    }
21
22    //buffer输出到屏幕
23    for (auto row = 0; row < rows; row++)
24    {
25        std::cout << buffer[row] << std::endl;
26    }
27
28    delete[] buffer;
29    std::cout << "-----G3" << std::endl;
30 }

```

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

```

1 void Draw3_vector(int num_x_per_cycle, int cycles, int coeff_y)
2 {
3     std::cout << "G3-vector-----" << std::endl;
4
5     int rows = num_x_per_cycle * cycles;
6     std::vector<std::string> buffer;
7     buffer.resize(rows);
8

```

```

9     int width = coeff_y * 2 + 4;
10    //buffer初始化, 背景清空成' '
11    for (auto row = 0; row < rows; row++)
12    {
13        buffer[row].resize(width, ' ');
14    }
15
16    //字符写入buffer
17    for (auto row = 0; row < rows; row++)
18    {
19        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
20        buffer[row][x] = '*';
21    }
22
23    //buffer输出到屏幕
24    for (auto row = 0; row <= rows; row++)
25    {
26        std::cout << buffer[row] << std::endl;
27    }
28
29    std::cout << "-----G3-vector" << std::endl;
30 }

```

此时, 可以比较便利地画图, 甚至画多个sin(x)

## G4-多图

```

1 //同时画多个图形
2 void Draw4(int num_x_per_cycle, int cycles, int coeff_y)
3 {
4     std::cout << "G4-----" << std::endl;
5
6     int rows = num_x_per_cycle * cycles;
7     std::vector<std::string> buffer;
8     buffer.resize(rows);
9
10    int width = coeff_y * 2 + 4;
11    //buffer初始化, 背景清空成' '
12    for (auto row = 0; row < rows; row++)
13    {
14        buffer[row].resize(width, ' ');
15    }
16
17    //字符写入buffer
18    for (auto row = 0; row < rows; row++)
19    {
20        int x = int(MyFun(row, num_x_per_cycle, coeff_y) + 0.5);
21        buffer[row][x] = '*';
22    }
23    for (auto row = 0; row < rows; row++)
24    {
25        int x = int(MyFun(row, num_x_per_cycle, coeff_y*0.5) + 0.5);
26        buffer[row][x] = '*';
27    }
28
29    //buffer输出到屏幕
30    for (auto row = 0; row < rows; row++)

```

```

31     {
32         std::cout << buffer[row] << std::endl;
33     }
34
35     std::cout << "-----G4" << std::endl;
36 }

```

## 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

```

1  struct RenderBuffer{
2      int                width;
3      int                height;
4      std::vector<std::string>    buffer;
5  };
6
7  struct RenderBuffer* InitBuffer(int width, int height)
8  {
9      RenderBuffer* rbuffer = new RenderBuffer;
10     rbuffer->width = width;
11     rbuffer->height = height;
12     rbuffer->buffer.resize(height);
13
14     for (auto y = 0; y < height; y++)
15     {
16         rbuffer->buffer[y].resize(width, ' ');
17     }
18     return rbuffer;
19 }
20
21 void ShowBuffer(struct RenderBuffer* rbuffer)
22 {
23     for (auto y = 0; y < rbuffer->height; y++)
24     {
25         std::cout << rbuffer->buffer[y] << std::endl;
26     }
27 }
28
29 void WriteBuffer(struct RenderBuffer* rbuffer, double x0, double y0, char c)
30 {
31     int x = int(x0 + 0.5);
32     int y = int(y0 + 0.5);
33
34     if (x < 0 || x >= rbuffer->width || y < 0 || y >= rbuffer->height)
35         return; //越界
36
37     rbuffer->buffer[y][x] = c;
38 }

```

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。

```

1 void Draw5(int num_x_per_cycle, int cycles, int coeff_y)
2 {
3     std::cout << "G5-----" << std::endl;
4
5     //注意：控制台窗口宽度有限制，会自动换行，所以设置小一些
6     //创建，并初始化
7     struct RenderBuffer* rbuffer = InitBuffer(100, 65);
8
9     //图形写到buffer
10    for (auto y = 0; y <= num_x_per_cycle * cycles; y++)
11    {
12        double x = MyFun(y, num_x_per_cycle, coeff_y);
13        writeBuffer(rbuffer, x, y, '*');
14    }
15
16    //buffer显示到屏幕
17    ShowBuffer(rbuffer);
18
19    delete rbuffer;
20
21    std::cout << "-----G5" << std::endl;
22 }

```

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

```

1 struct RenderBuffer_single_string {
2     int width;
3     int height;
4     std::string buffer;
5 };
6
7 void InitBuffer(int width, int height, struct RenderBuffer_single_string&
8 rbuffer)
9 {
10    rbuffer.width = width;
11    rbuffer.height = height;
12    rbuffer.buffer.resize(height * (width + 1), ' ');
13    for (auto y = 0; y < height; y++)
14    { //每隔width个字符设置一个换行符'\n'
15        rbuffer.buffer[y * (width + 1) + width] = '\n';
16    }
17 }
18
19 void ShowBuffer(const struct RenderBuffer_single_string& rbuffer)
20 {
21    std::cout << rbuffer.buffer << std::endl;
22 }
23
24 void writeBuffer(struct RenderBuffer_single_string& rbuffer, double x0,
25 double y0, char c)
26 {

```

```

25     int x = int(x0 + 0.5);
26     int y = int(y0 + 0.5);
27     if (x < 0 || x >= rbuffer.width || y < 0 || y >= rbuffer.height)
28         return; //越界
29     rbuffer.buffer[y * (rbuffer.width + 1) + x] = c;
30 }

```

调用时

```

1 void Draw6(int num_x_per_cycle, int cycles, int coeff_y)
2 {
3     std::cout << "G6-----" << std::endl;
4
5     //创建buffer
6     struct RenderBuffer_single_string rbuffer;
7     //初始化buffer
8     InitBuffer(100, 65, rbuffer);
9
10    //字符写到buffer
11    for (auto y = 0; y <= num_x_per_cycle * cycles; y++)
12    {
13        double x = MyFun(y, num_x_per_cycle, coeff_y);
14        writeBuffer(rbuffer, x, y, '*');
15    }
16
17    //buffer显示到屏幕
18    ShowBuffer(rbuffer);
19
20    std::cout << "-----G6" << std::endl;
21 }

```

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

```

1 void Draw7(int num_x_per_cycle, int cycles, int coeff_y)
2 {
3     std::cout << "G7-----" << std::endl;
4     //创建buffer
5     struct RenderBuffer_single_string rbuffer;
6     //初始化buffer
7     InitBuffer(100, 65, rbuffer);
8
9     //竖着的sin
10    for (auto y = 0; y <= num_x_per_cycle * 0.8 * cycles; y++)
11    {
12        double x = MyFun(y, num_x_per_cycle * 0.8, coeff_y);
13        writeBuffer(rbuffer, x, y, '*');
14    }
15    //横着的sin
16    for (auto x = 0; x <= num_x_per_cycle * cycles; x++)
17    {

```

```

18     double y = MyFun(x, num_x_per_cycle, coeff_y * 0.8);
19     WriteBuffer(rbuffer, x, y, '*');
20 }
21
22 //buffer显示到屏幕
23 ShowBuffer(rbuffer);
24
25 std::cout << "-----G7" << std::endl;
26 }

```

### 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

```

1  class CRenderBuffer
2  {
3  private:
4      int          width;
5      int          height;
6      std::string  buffer;
7  public:
8      void Init(int width0, int height0)
9      {
10         width = width0;
11         height = height0;
12         buffer.resize(height * (width + 1), ' ');
13         for (auto y = 0; y < height; y++)
14             { //每隔width个字符设置一个换行符'\n'
15                 buffer[y * (width + 1) + width] = '\n';
16             }
17     }
18
19     void Show()
20     {
21         std::cout << buffer << std::endl;
22     }
23
24     void Write(double x0, double y0, char c)
25     {
26         int x = int(x0 + 0.5);
27         int y = int(y0 + 0.5);
28         if (x < 0 || x >= width || y < 0 || y >= height)
29             return; //越界，不画
30         buffer[y * (width + 1) + x] = c;
31     }
32 };

```

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。



```

1 void Draw8(int num_x_per_cycle, int cycles, int coeff_y)
2 {
3     std::cout << "G8-----" << std::endl;
4     //创建CRenderBuffer的对象rbuffer
5     CRenderBuffer rbuffer;
6     //初始化buffer, 通过对象rbuffer发起方法调用, 初始化buffer
7     rbuffer.Init(100, 65);
8
9     for (auto y = 0; y <= num_x_per_cycle * cycles; y++)
10    {
11        double x = MyFun(y, num_x_per_cycle, coeff_y);
12        rbuffer.Write(x, y, '*');
13    }
14
15    rbuffer.Show();
16    std::cout << "-----G8" << std::endl;
17 }

```

## 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。

```

1 class IRenderBuffer
2 {
3     protected:
4         int          width;
5         int          height;
6     public:
7         virtual void Init(int width0, int height0)=0;
8         virtual void Show() = 0;
9         virtual void Write(double x0, double y0) = 0;
10 };

```

然后派生出具体的实现子类, 比如之前例子所用的方法

```

1 class CRenderBuffer : public IRenderBuffer
2 {
3     private:
4         std::string  buffer;
5     public:
6         void Init(int width0, int height0)
7         {
8             width = width0;
9             height = height0;
10            buffer.resize(height * (width + 1), ' ');
11            for (auto y = 0; y < height; y++)
12                //每隔width个字符设置一个换行符'\n'
13                buffer[y * (width + 1) + width] = '\n';
14        }
15    }
16
17    void Show()

```

```

18     {
19         std::cout << buffer << std::endl;
20     }
21
22     void Write(double x0, double y0)
23     {
24         int x = int(x0 + 0.5);
25         int y = int(y0 + 0.5);
26         if (x < 0 || x >= width || y < 0 || y >= height)
27             return; //越界, 不画
28         buffer[y * (width + 1) + x] = '*';
29     }
30 };

```

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

```

1 void Draw9(IRenderBuffer &buffer, int num_x_per_cycle, int cycles, int
  coeff_y)
2 {
3     std::cout << "G9-----" << std::endl;
4
5     //初始化buffer, 通过对象rbuffer发起方法调用, 初始化buffer
6     buffer.Init(100, 65);
7
8     for (auto y = 0; y < num_x_per_cycle * cycles; y++)
9     {
10         double x = MyFun(y, num_x_per_cycle, coeff_y);
11         buffer.Write(x, y);
12     }
13     buffer.Show();
14     std::cout << "-----G9" << std::endl;
15 }

```

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

```

1 class MRGB
2 {
3 public:
4     int R;
5     int G;
6     int B;
7 public:
8     MRGB()
9     {
10         R = G = B = 0;
11     }
12     MRGB(int r, int g, int b)
13     {
14         R = r, G = g, B = b;
15     }
16 };
17
18 class CRenderBuffer_image : public IRenderBuffer
19 {

```

```

20 private:
21     std::vector<struct MRGB>    buffer;
22 public:
23     void Init(int width0, int height0)
24     {
25         width = width0;
26         height = height0;
27         buffer.resize(width * height);
28     }
29
30     void Show()
31     {
32         FIBITMAP* bitmap = FreeImage_Allocate(width, height, 24);
33         if (!bitmap)
34         {
35             return;
36         }
37
38         for (int y = 0; y < height; y++)
39         {
40             for (int x = 0; x < width; x++)
41             {
42                 MRGB mrgb = buffer[y * width + x];
43                 RGBQUAD rgb;
44                 rgb.rgbRed = mrgb.R, rgb.rgbGreen = mrgb.G, rgb.rgbBlue =
mrgb.B;
45                 FreeImage_SetPixelColor(bitmap, x, y, &rgb); //设置像素值
46             }
47         }
48
49         //存图像
50         std::string file = "result.jpg";
51         FreeImage_Save(FIF_PNG, bitmap, file.c_str());
52         FreeImage_Unload(bitmap); //释放FreeImage
53     }
54
55     void write(double x0, double y0)
56     {
57         int x = int(x0 + 0.5);
58         int y = int(y0 + 0.5);
59         if (x < 0 || x >= width || y < 0 || y >= height)
60             return; //越界, 不画
61
62         buffer[y * width + x] = MRGB(0, 255, 0);
63     }
64 };
65

```

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。

```

1  int main()
2  {
3      int num_x_per_cycle = 30;
4      int cycle = 2;
5      int coeff_y = 20;
6
7      { //过去的方法

```

```
8     CRenderBuffer rbuffer;
9     Draw9(rbuffer, num_x_per_cycle, cycle, coeff_y);
10    }
11    {//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
12     CRenderBuffer_image imageBuffer;
13     Draw9(imageBuffer, num_x_per_cycle, cycle, coeff_y);
14    }
15 }
```