

C++语言疯狂实践系列

雷小锋 编著

中国矿业大学计算机学院

前 言

计算机语言教学必须围绕问题求解的终极目标展开，培养自顶向下、层层分治的计算思维。否则即使掌握了大量的语法工具，面对实际问题仍然是有心无力，无处下手。

（1）以问题求解需求为导向构建 C++语言宏观知识结构。

C++语言不仅是形式繁复的一堆语法工具，更是问题求解需求的一组表达工具。学习 C++语言乃至任何计算机语言，不是摆弄纷繁复杂的语法工具和语法技巧，而是梳理人类求解问题思维方式和表达需求，然后在问题求解需求的框架下建立 C++语言众多语法工具的结构和联系，以问题求解需求为导向理解语法工具的表达方法。

因此，本书第一章就从人类的三种问题求解思维出发，以问题求解需求为导向组织梳理 C++语言的宏观知识结构，纵览 C++语言全局。

（2）通过复杂问题求解实践实现 C++语言的融会贯通。

在 C++语言的教学过程和实践训练当中，提供的编程案例通常是一些玩具问题，主要目标是验证说明特定语法工具的使用方法和禁忌，有利于学习者迅速掌握某种单一的语法工具。同时，也会导致语言学习者缺乏综合性问题求解实践的训练，不能完成语法工具的融合贯通，无法建立自顶向下、层层分治的计算思维，缺乏问题求解方法的经验积累，在面对稍具规模的复杂问题时便感觉有心无力，无处下手。

因此，本书在第一章 C++语言全局纵览之后，便由浅入深提供一系列较为复杂的编程案例。在每个编程案例中，除了给出最终实现的程序代码之外，更重要的是把完美的代码撕碎开来，呈现出程序编写循序渐进的思维分析推理过程、从问题求解需求到语法工具选择的因果链路、以及复杂问题层层分解的分治计算思维。

C++语言的宏大体系决定了学习者必然要经历长期艰苦的修炼过程。以问题求解需求为导向构建 C++语言知识结构，领悟语法工具的形式和内涵，是万里长征第一步。要熟练运用 C++语言实现问题求解的终极目标，必须经过疯狂实践训练的磨砺。

西方谚云：“Practice makes perfect”。

卖油翁说：“万事无他，唯手熟尔”。

作 者

2018 年 9 月

目 录

C++语言全局纵览	1
1. 过程化思维与过程化编程	1
1.1 描述数据	1
1.2 描述数据处理过程	6
1.3 结构化编程与程序组织	9
2. 面向对象思维与面向对象编程	11
2.1 描述对象	11
2.2 对象的组合与继承	17
2.3 异常处理	20
3. 泛型思维与泛型编程	20
3.1 泛化数据类型：定义模板	20
3.2 应用环节：模板实例化	21
3.3 C++标准模板库	21
疯狂实践系列一：素数判定问题	25
1. 训练目标	25
2. 准备编程环境	25
2.1 新建一个工程项目（Project）	26
2.2 C++程序的固有结构	27
3. 编写素数判定程序	29
3.1 利用注释描述问题求解的框架	29
3.2 把注释翻译为 C++代码	30
3.3 尝试程序编译链接	32
3.4 添加 isprime 函数的定义	33
3.5 程序编译链接和执行	37
4. 完整的程序代码	38
疯狂实践系列二：自身和反序数均是素数	41

1.	训练目标	41
2.	编写问题求解的代码框架	41
3.	翻译任务 1: 让 <code>i</code> 从 10 变到 1000。	41
4.	翻译任务 2: 判定 <code>i</code> 及其反序数是否素数	42
5.	定义 <code>bothIsPrime</code> 函数	43
5.1	定义 <code>invert</code> 函数	44
5.2	定义 <code>isprime</code> 函数	45
6.	完整的程序代码	45
疯狂实践系列三: 最大公约数和最小公倍数		47
1.	训练目标	47
2.	编写问题求解的代码框架	47
3.	定义 <code>gcd</code> 函数	48
3.1	暴力测试法实现 <code>gcd</code>	48
3.2	欧几里德算法实现 <code>gcd</code>	48
3.3	欧几里德算法递归实现 <code>gcd</code>	49
疯狂实践系列四: 进制转换 10to2		51
1.	训练目标	51
2.	自顶向下分治	51
2.1	第一层子问题: <code>dec2bin</code> 函数	52
2.2	第二层子问题: <code>printhead</code> 、 <code>printitem</code> 、 <code>printfoot</code> 函数	53
2.3	第三层子问题: <code>printtab</code> 、 <code>printsep</code> 函数	54
3.	函数集中声明	55
4.	头文件集中声明	56
4.1	定义头文件	56
4.2	包含头文件	57
4.3	预编译头文件	58
疯狂实践系列五: 谁参加了会议		61
1.	训练目标	61
2.	问题求解思路: 构造解空间	61

3.	自顶向下分治	62
3.1	第一层问题: <code>main</code> 函数	62
3.2	第二层问题: 定义 <code>IsAnswer</code> 和 <code>print</code> 函数	62
3.3	最后的堡垒: 定义 <code>join</code> 函数	64
3.4	完整代码组织结构	65
疯狂实践系列六: 约瑟夫问题之还有谁		67
1.	训练目标	67
2.	自顶向下分治	67
2.1	第一层问题: <code>main</code> 函数	67
2.2	难啃的骨头: 定义 <code>play</code> 函数	69
3.	通过调试定位 <code>BUG</code>	72
3.1	朴素调试技术: 打印输出	72
3.2	使用断言技术	75
3.3	使用调试工具	77
4.	更高效的问题求解思路	82
疯狂实践系列七: 螺旋输出 $1 \sim N^2$		85
1.	训练目标	85
2.	自顶向下分治	85
2.1	第一层问题: <code>main</code> 函数	85
2.2	第二层问题: 定义 <code>printSquare</code> 函数	86
2.3	第二层问题: 定义 <code>printSpiral</code> 函数	86
3.	换个思路: 遍历地图	89
3.1	重新定义 <code>printSpiral</code> 函数	89
3.2	条条大路通罗马	92
疯狂实践系列八: 排座座吃果果		93
1.	训练目标	93
2.	自顶向下分治	93
2.1	第一层问题: <code>main</code> 函数	93
2.2	第二层问题: 数据的产生和读写	94

2.3	第二层问题：选择排序及 <code>xsort</code> 函数	95
3.	实现从大到小排序	97
4.	通用排序函数 <code>gsort</code>	98
5.	完整代码组织结构	98
疯狂实践系列九：日期问题		101
1.	训练目标	101
2.	自顶向下分治	101
2.1	第一层问题： <code>main</code> 函数	101
2.2	第二层问题：定义 <code>CDate</code> 类	102
3.	单元测试保障代码质量	109
3.1	创建一个 C++单元测试项目	109
3.2	在单元测试项目中引入被测试代码	110
3.3	编写单元测试代码	110
3.4	执行单元测试	112
4.	完整代码组织结构	113
疯狂实践系列十：最短路径问题		115
1.	训练目标	115
2.	自顶向下分治	115
2.1	第一层问题： <code>main</code> 函数	115
2.2	第二层问题： <code>buildMap</code> 函数	117
2.3	第二层问题：定义 <code>Map</code> 类	117
2.4	暴力穷举计算最短路径： <code>getShortestPath</code> 函数	121
2.5	完整的代码组织结构	129
3.	更高效的 <code>Dijkstra</code> 算法	131
3.1	<code>Dijkstra</code> 算法思路	131
3.2	<code>Dijkstra</code> 算法 <code>main</code> 函数	133
3.3	定义 <code>Map</code> 类	133
疯狂实践系列十一：方程求根问题		139
1.	训练目标	139

2.	自顶向下分治	139
2.1	求解问题需要哪些对象	139
2.2	第一层问题: <code>main</code> 函数	140
2.3	第二层问题: 定义 <code>Equation</code> 类	141
2.4	第二层问题: 初步定义 <code>Solver</code> 类	142
3.	非线性方程求解方法	144
3.1	等间隔搜索法	145
3.2	二分搜索法	145
3.3	牛顿迭代法	145
3.4	弦截法	146
4.	第三层问题: 派生出具体方程求解器	146
4.1	重新设计 <code>Solver</code> 类	146
4.2	派生出四种方程求解器	148
5.	完整代码组织结构	150
疯狂实践系列十二: 迷你计算器		153
1.	训练目标	153
2.	自顶向下分治	153
2.1	第一层问题: <code>main</code> 函数	153
2.2	表达式求值的原理和过程	154
2.3	推断 <code>Expression</code> 类的需求	157
3.	第二层问题: 定义 <code>Expression</code> 类	158
3.1	重载 <code>Expression</code> 的输出运算符	159
3.2	定义 <code>Expression</code> 类的构造函数	160
3.3	兑现假设: 定义 <code>Element</code> 类	170
3.4	计算求值: 定义 <code>calc</code> 函数	171
4.	完整代码组织结构	173
疯狂实践系列之编程无极限		175
参考文献		177

C++语言全局纵览

人类的问题求解思路，用计算机语言描述出来，就是**程序**（program），描述的过程就是编写程序，简称**编程**（programming）。计算机根据程序的指示按部就班的工作，就是**执行程序**，通过执行程序我们就可以得到问题求解的最终结果。

程序是人类的问题求解思路，人类有什么样的问题求解思维，计算机语言就应该提供相应的语法工具来支持问题求解思维的描述。针对过程化、面向对象、泛型三种问题求解思维，C++语言提供了过程化、面向对象、泛型编程的系列语法工具。

1. 过程化思维与过程化编程

过程化问题求解思维认为世界是事物状态发展变化的过程，求解问题就是对事物进行加工处理，促使其状态发生一系列变化，从而达到预期的目标状态。

过程化编程，就是在过程化思维下描述人的问题求解思路。因此，过程化编程，就需要描述数据以及数据的加工处理过程，包括：

- （1）**描述数据**：说明求解问题需要哪些数据，这些数据具有什么样的规格。
- （2）**描述数据处理过程**：说明数据如何输入、如何加工处理、以及如何输出处理结果的过程，包括输入数据、处理数据和输出结果三个环节。

1.1 描述数据

描述数据，就是告诉计算机关于待处理数据的一些规格信息，包括：

- （1）数据在存储器中占据多大的存储空间。
- （2）数据如何转化成存储器要求二进制形式（编码方式）。
- （3）如何把存储器中二进制形式恢复成数据（解码方式）。
- （4）数据可以进行哪些运算处理。

1.1.1 数据类型：约定数据规格信息

C++语言通过**数据类型**的概念来约定数据的规格信息。在 C++语言中，提供了一系列预先约定的数据类型，主要包括三类（如图 1 所示）。

- (1) 内置数据类型：描述常用的整型、字符型、浮点型数据。
- (2) 复合数据类型：描述常变量、数组、指针、引用等复合数据。
- (3) 定制数据类型：定制自己的数据类型，建立我们自己关于存储空间大小、编码和解码方式、及其可接受运算处理行为的数据规格约定。

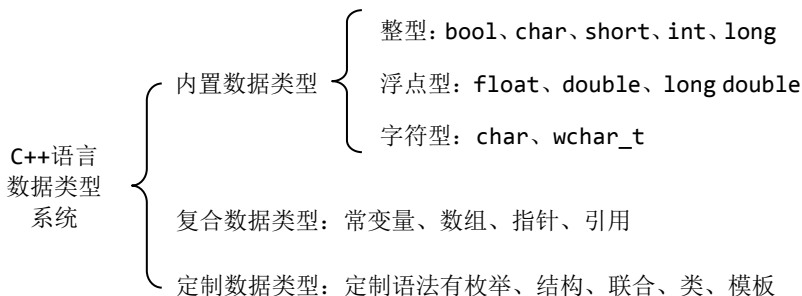


图 1 C++语言的数据类型系统

1.1.2 描述不变数据：字面量

根据形式和用途，C++语言提供了整型、字符型、浮点型、布尔型、字符串型五种字面量。不同类型的字面量有不同的构造方式。

- (1) 整型字面量：157、-0655、0xFE、-0x1UL。
- (2) 字符型字面量：'C'、'\$'、'\n'、'\x61'、L'学'、L'a'。
- (3) 浮点型字面量：0.2、0.2f、0.2E-01。
- (4) 布尔型字面量：true、false。
- (5) 字符串型字面量："英语学习"、L"ABC"、"x41\x42\x43"。

1.1.3 描述可变数据：变量

变量是约定了名字和数据类型的存储空间，通俗讲就是放置数据的容器。

变量 = 变量名 + 数据类型 = 数据地址 + 数据类型。

通过变量的名字我们可以方便地对数据进行存取访问。

- (1) 变量如何定义：数据类型 + 名字。如何给变量取名字。
- (2) 变量如何初始化：赋值语法、构造函数语法。
- (3) 访问变量：访问变量的容器、访问变量中的数据值、访问变量的地址。

1.1.4 描述复合数据

在内置数据类型的基础上，C++语言引入复合数据类型。对于这些复合数据类型的变量，我们需要关注：该类型变量的用途和适用场合、如何定义和初始化变量、变量的存储空间映像、如何使用这类变量。

(1) 常变量：又称符号常量，描述内容不可修改的变量，用于替代字面量。

```
const double PI = 3.142; //或者
const double PI(3.142); //或者
double const PI = 3.142; //交换 const 和 double 的次序
```

(2) 数组变量：用于描述类型相同的一组变量，存储相同类型的一组数据。

- ✧ 定义数组变量：元素类型数组变量名[元素个数]。
- ✧ 设定数组元素的个数：整型常量。
- ✧ 数组的存储空间映像： $\text{sizeof}(\text{数组名}) = \text{sizeof}(\text{数组类型})$
 $= \text{sizeof}(\text{数组元素类型}) * \text{数组元素个数}$ 。
- ✧ 计算数组元素的地址：表达式中数组变量名求值为数组首元素地址。
- ✧ 初始化数组变量：初始化列表{ }。
- ✧ 访问数组元素：下标运算，越界访问。
- ✧ 多维数组：元素是数组的数组变量。
- ✧ 多维数组的访问：多个下标运算。

(3) 指针变量：用于存放其他变量的地址，通过指针间接操作其他的变量。

- ✧ 定义指针变量：数据类型 * 指针名。
- ✧ 指针变量的存储空间映像：指针变量和指针所指变量的关系。
- ✧ 指针的访问和操作：访问指针变量的容器、内容和地址。
- ✧ 访问指针所指变量：解引用运算。
- ✧ 指针变量与整数的加法和减法运算：求值结果是地址。
- ✧ 指针与数组：指针执行数组首元素，指针就会变身数组。
 访问数组元素：数组+下标、数组+解引用、指针+解引用、指针+下标。
- ✧ 无类型指针：可以存放任何类型变量的地址。
- ✧ 指针与动态变量。

定义动态变量: `数据类型 * 指针名 = new 数据类型(初始值);`

销毁动态变量: `delete` 指向动态变量的指针;

✧ 指针与动态数组。

定义动态数组: `元素类型 * 指针变量名 = new 元素类型[元素个数];`

销毁动态数组: `delete[]` 指向动态数组首元素的指针;

✧ 设定动态数组的元素个数: 允许通过变量设定。

(4) 字符串变量。

✧ 字符数组实现字符串: `char str[] = "Hi, Stop";`

✧ 字符指针实现字符串: `char *pStr = "Hi, Stop";`

✧ 字符串操作函数: `strlen`、`strcpy`、`strcat`、`strcmp`、`strstr`

✧ `string` 类: `#include <string>`

(5) 引用变量: 用于给其他变量取一个的别名, 通过引用操作其他变量。

✧ 引用定义: `数据类型&引用名 = 目标变量;`

✧ 引用作为函数参数: 两种作用 (a) 通过修改形参可以实现对实参的修改; (b) 与实参变量共享其存储空间, 节省开销。

(6) 常量数组, 数组元素是常变量的数组。

```
const int week[] = {0, 1, 2, 3, 4, 5, 6};
enum {Sun, Mon, Tue, Wen, Thur, Fri, Sat}; //枚举更好
```

(7) 指针数组, 数组元素是指针变量的数组。

```
double* *pdpAry = new double*[5]; //动态数组必须 delete[]
```

(8) 数组指针, 指向数组的指针。

```
double* (*aryPointer)[5] = &pdpAry;
```

(9) 常量指针: 指向常变量的指针。

```
const double PI = 3.142;
const double *pConst = &PI;
```

(10) 指针常量: 指针型的常变量。

```
double dVal = 1.1;
double * const cpVal = &dVal; //不可修改的指针
```

(11) 指向常量的常指针。

```
const double PI = 3.142;
```

```
const double * const cpVal = &PI; //指向常变量的常指针
```

(11) 多维数组：数组元素是数组的数组。

(12) 多级指针：指向指针的指针。

```
int iMyAge = 30, *pMyAge = &iMyAge;
int* *ppMyAge = &pMyAge;
int** * pppMyAge = &ppMyAge;
```

(13) 指针引用：给指针起的别名。

```
int**p2d; //二级指针 p2d
int** &refPtr= p2d;
```

(14) 数组引用：给数组起的别名。

```
inta2d[3][50]; //二维数组 a2d
int (&refAry)[3][50] = a2d;
```

(15) 常量引用：给常变量起的别名。

```
const double PI = 3.142;
const double &refPI = PI;
```

(16) 简化嵌套复合：typedef。

```
typedef int* IntPtr;
typedef IntPtr* IntPtr2rd; //二级指针
typedef IntPtr2rd &IntPtr2rdRef; //二级指针的引用
typedef double* AryOfDb1Ptr[5]; //指针数组
```

1.1.5 复合数据与函数

(1) 数组用作函数参数

- ✧ C++语言约定：数组用作函数形参时会退化为数组元素类型的指针，数组元素的个数信息就会损失掉。
- ✧ 此时需要给函数增加一个专门的形式参数，用来传递形参数组的元素个数。

```
int max(int numbers[], int N);
```

(2) 函数如何返回数组名

数组名求值为数组首元素的地址，返回数组名其实是返回指向数组首元素的指针。

C++语言不允许数组作为函数的返回类型，而是用指针替代。

(3) 指针函数

指针函数，等同于函数返回数组名。函数返回数组名，实际上返回了指向数组首元素的指针，故指针可以作为函数的返回类型。

(4) 函数指针：指向函数的指针，通过函数指针可以间接调用函数

- ✧ 函数的入口地址：C++语言约定，函数名代表了函数的入口地址。
- ✧ 函数的类型信息：由函数的形式参数类型及其返回类型来共同约定。
- ✧ 定义函数指针

```
int max(int numbers[], int N);    //查找最大元素
int (*pMax) (int[], int)= max;
```

- ✧ 调用函数指针。

```
int nums[] = {25, 32, 41, 5, 78, 44, 31};
int pos = pMax(nums, sizeof(nums)/sizeof(int));
```

(5) 函数指针的数组；

- ✧ 定义函数指针的数组；

```
int max(int numbers[], int N);    //查找最大元素
int min(int numbers[], int N);    //查找最小元素
int absmax(int numbers[], int N); //查找绝对值最大元素
int absmin(int numbers[], int N); //查找绝对值最小元素
int (*findelem[])(int[], int) = {max,min,absmax,absmin};
```

- ✧ 通过函数指针的数组批量调用函数；

```
int nums[] = {25, 32, 41, 5, 78, 44, 31};
for(int i=0; i<sizeof(findelem)/sizeof(FINDELEM); i++)
    int pos = findelem[i](nums, sizeof(nums)/sizeof(int));
```

- ✧ 函数指针用作函数参数，实现通用函数。

1.2 描述数据处理过程

数据的加工处理过程，包括最基本的运算处理，复杂一点的流程控制，数据输入输出处理、以及把很多加工处理过程打包封装起来的函数。

1.2.1 基本的运算处理：运算符和表达式

表达式，是一个或多个操作数通过**运算符**连接而成的运算序列，用来描述一个简单的运算处理过程。C++语言中表达式的功能有两个：

(1) 运算求值产生结果：求值结果是左值或者右值。其中，左值表示一个可解释的存储空间，右值表示一个临时的计算值。

(2) 产生副作用：即表达式在运算求值之外的作用，修改变量就是一种副作用。

表达式需要约定操作数和运算符绑定或者结合的先后顺序，C++语言通过运算符的**优先级**和**结合性**来规定表达式的结合顺序。其中，优先级高的运算符先结合，优先级相同时，则按照结合性从左向右或者从右向左进行结合。

注意，优先级和结合性只是规定运算符和操作数的结合次序，而非求值次序，求值次序通过时序点来保证。对于多个并列的子表达式，C++语言通常不约定子表达式的求值顺序，逻辑与&&、逻辑或||、条件运算符?:、以及逗号运算符等是例外。

当参与表达式运算的多个操作数的数据类型不一致时，会发生类型转换。C++语言提供**隐式类型转换**和**显式类型转换**两种方式。

在 C++语言中，提供了丰富的**运算符**（又称**操作符**），用于构造各种类型的表达式，描述针对不同类型数据所进行的运算处理。

- (1) 算术运算：加 (+)、减 (-)、乘 (*)、除 (/)、取余 (%)、++、--
- (2) 逻辑运算：逻辑非 (!)、逻辑与 (&&)、逻辑或 (||)
- (3) 关系运算：等于 (==)、>、>=、<、<=、不等于 (!=)
- (4) 位串运算：位非 (~)、位与 (&)、位或 (|)、位异或 (^)、左移 (<<)、右移 (>>)
- (5) 赋值运算：=、+=、-=、*=、/=、%=、&=、|=、<<=、>>=、^=
- (6) 其他运算：.、->、[]、()、.*、->*、typeid、const_cast、dynamic_cast、reinterpret_cast、static_cast、sizeof、new、delete、,、?:

1.2.2 控制数据处理流程

C++语言提供了选择、循环和跳转三类语句，用于控制程序的流程逻辑。

- (1) 选择（“如果……就”）：if 语句、switch 语句。
- (2) 循环（“如果……就反复执行”）：while、do-while、for 循环语句。
- (3) 跳转：break、continue、return、goto。

1.2.3 数据输入输出处理

数据输入，指的是将外部输入设备中的数据放入存储器；而数据输出，则指的是将存储器中的数据写入外部的输出设备。C++语言通过“流”来处理输入输出。

- (1) 标准输入流：绑定到键盘的输入流对象 `cin`，支持键盘输入；
- (2) 标准输出流：绑定到显示器的输出流对象 `cout`，支持屏幕输出；
- (3) 文件输入流 `ifstream`：绑定到特定磁盘文件，支持读文件；
- (4) 文件输出流 `ofstream`：绑定到特定磁盘文件，支持写文件；
- (5) 文件输入输出流 `fstream`：绑定到特定磁盘文件，支持读写文件；

可以通过设置格式标记或使用格式操作子，控制输入输出的数据格式。

1.2.4 制造数据处理机器：函数

函数，是将数据处理过程封装成逻辑独立、可重用的语法单元，相当于制造一个数据处理机器。

- (1) 定义函数，就是制造数据处理机器，需要说明函数四要素：

返回类型 函数名(形式参数表){函数体}

- ✧ 函数名，代表函数机器的名字，必须符合标识符的构词规则。
- ✧ 形式参数，简称形参，表示将来会输入函数机器的待处理数据。
- ✧ 返回类型，函数机器运转产生的结果数据需要通过 `return` 语句反馈回去，称为**返回值**，返回值的数据类型，简称返回类型。
- ✧ 函数体，表示函数机器运转的过程，描述函数所封装的数据及数据处理过程。

- (2) 调用函数：就是使用数据处理机器，指定机器名和实际参数：

函数名(实参列表)

- ✧ 实参，即实际参数，是在使用数据处理机器时指定的实际要处理的数据。
- ✧ 形参，即形式参数，是在制造数据处理机器时指定的未来会存在的数据。

- (3) 函数声明

C++语言允许函数先调用后定义，但是，它要求在调用函数之前，必须进行函数声明以保证函数的存在性。定义包含声明，一次定义，多次声明。

- ✧ 函数定义 = 返回类型 + 函数名 + 形参表 + 函数体
- ✧ 函数声明 = 返回类型 + 函数名 + 形参表

(4) 函数调用的执行流程：实参求值->流程跳转->形参生成->被调函数执行->流程回转->主调函数重启。

(5) 形参生成：调用函数，需要根据实参产生形参。两种形参变量的生成方式：

- ✧ 克隆方式：根据实参克隆出形参，形参的变化不影响实参；
- ✧ 别名方式：形参是实参的别名，修改形参就是修改实参；

(6) 函数的嵌套调用和递归调用

(7) 函数内联：函数体在调用点展开，节省函数调用的开销。

(8) 函数重载：函数的名字可以重复利用，但必须保证形式参数不同，即函数重载。在函数调用时，编译器会选择与实参类型和个数匹配最好的重载函数来调用。

(9) 函数的默认参数值：通过函数声明设定参数的默认值。

1.3 结构化编程与程序组织

通过函数，可以比较轻松地应付规模较大问题的编程求解。在求解较大问题时，如果遇到其中一些环节或者子问题暂时难以处理，此时，可以通过函数声明，假设存在某个函数可以解决这些子问题，即使它们暂时还不存在；然后通过调用这个函数完成问题的整体编程求解。之后，再在合适的时机给出这个函数的定义。

自上而下将复杂问题不断细分，产生一系列更易于求解的子问题，然后再针对这些子问题进行编程求解的思维，就是**结构化编程思想**，或者说，自顶向下、逐步求精的编程思想，或者分治计算思维。

1.3.1 函数声明的组织方式

(1) 就地声明，非常适合于编写规模较小的“玩具”程序。

(2) 在源文件开始处集中声明。

(3) 通过头文件集中声明。

- ✧ 定义头文件，生成一个独立的文件，文件扩展名通常为*.h，在其中放置函数或者变量的声明。
- ✧ 包含头文件。如果在编写程序时需要声明头文件中的函数，则只需要把该头文件通过#include 预编译指令包含进源文件中即可。

```
#include <头文件名>    //包含标准头文件
```

```
#include "头文件名"    //包含用户头文件
```

✧ 避免头文件重复包含：利用**#ifndef**、**#endif** 和**#define** 进行条件编译。

1.3.2 程序的多文件组织结构

头文件以及独立编译机制，非常有利于团队协作进行程序开发，也导致了将程序文件按功能逻辑进行分割和组织的多文件程序组织结构。为了避免多文件之间程序实体的冲突以及共享，C++语言约定了一系列的语法机制，包括：

(1) 作用域：程序实体存在的空间范围，包括局部域、名字空间域、类域、以及文件域。其中，局部域有复合语句的块域、函数域、以及函数声明域。

(2) 生存期：程序实体存在的时间范围，包括静态、自动和动态生存期。C++语言允许在不同的作用域定义不同生存期的变量。

✧ 局部域+静态生存期：局部静态变量

✧ 局部域+自动生存期：自动变量

✧ 文件域+静态生存期：全局变量

✧ 作用域+动态生存期：动态变量

(3) 链接属性：多文件之间程序实体的保护和共享；

✧ 内部链接性：**static** 修饰，不允许共享文件域中的程序实体。

✧ 外部链接性：**extern** 修饰，允许程序实体延伸到所在文件域的外部。

(4) 名字空间：约定了一个命名的作用域，并将其成员隐藏在该作用域中，使其对外不是直接可见的，从而避免了全局程序实体与外界发生冲突。

✧ 定义名字空间：**namespace** 名字空间的名字{ }。

✧ 访问名字空间成员：名字空间的名字::成员名。

✧ 使用头文件引入声明。

✧ 使用 **using** 指示符：

```
using 名字空间名::成员名;  
using namespace 名字空间名;
```

✧ C++标准名字空间：**std**。

2. 面向对象思维与面向对象编程

面向对象问题求解思维认为世界是由一系列对象组成的，每种类型的对象都具有自身的属性和行为能力。找到一组合适的对象，通过对象之间的沟通协作发挥各自的行为能力，就可以实现问题求解。

面向对象编程，就是在面向对象思维下描述人的问题求解思路。因此，面向对象编程，就需要描述问题求解的对象以及对象之间的沟通协作过程：

(1) **描述对象**：说明求解问题需要哪些对象，这些对象应该具有什么样的规格要求（包括具有哪些属性和行为能力）；

(2) **描述沟通协作过程**：说明如何产生具体的对象，以及这些对象之间如何沟通协作发挥各自的行为能力实现问题求解的过程。

对象之间的沟通协作过程，可以利用过程化编程的语法工具来描述。面向对象编程的核心工作是描述对象以及如何产生具体对象并发挥对象的行为能力。

2.1 描述对象

描述对象就需要完成两方面的工作：(a) **定义类**，描述一类对象共有的属性和行为特征；(b) **产生对象**，根据类生成具体的对象实例。

2.1.1 定义类

类描述了具有相同属性和行为能力的一类对象，称为对象类型，简称类。在类中，对象的属性称为类的**数据成员**，对象的行为能力称为类的**成员函数**。

(1) 定义类的语法

```
class 类名 {  
    //1. 数据成员：描述类的属性特征，变量语法（数据类型+名字）  
    //2. 成员函数：描述类的行为特征，函数语法  
}; //注意结尾的分号
```

(2) 类成员封装：三种访问限定符。

✧ **private** 修饰的类成员，私有成员，只有在类自身的成员函数中才能访问对象的私有成员；

- ✧ **public** 修饰的类成员，公有成员，在任何函数（包括成员函数和非成员函数）中都可以访问对象的公有成员；
- ✧ **protected** 修饰的类成员，保护成员，可以在类自身成员函数及其派生类的成员函数中访问对象的保护成员。

（3）在类体中定义成员函数，在类体外部定义成员函数，成员函数的内联。

（4）类是定制的数据类型，约定了一类对象应该具有的属性 and 行为特征，同时约定了该类对象的存储空间大小、编码和解码方式以及可以接受的运算和处理行为。

```
类 = 属性特征 + 行为特征
    = 数据成员 + 成员函数
    = 公有成员 + 私有成员
```

2.1.2 产生对象

（1）产生对象的语法，与产生变量类同。

```
变量 = 名字 + 数据类型 = 地址 + 数据类型
对象 = 名字 + 类 = 地址 + 类
```

（2）访问对象成员。

- ✧ 在外部函数中访问对象成员：只能访问对象的公有成员，可以使用点号或箭头语法进行访问。

```
对象名.成员名
对象指针->成员名
```

- ✧ 在类成员函数中访问另一个同类对象的成员：可以访问另一个同类对象的公有、私有和保护成员，可以使用点号或箭头语法进行访问。
- ✧ 在类成员函数中访问自身对象的成员

```
成员名
this->成员名
```

2.1.3 类成员：约定对象的属性和行为

（1）类的数据成员：约定对象的属性特征。

- ✧ 常量数据成员：**const** 修饰的数据成员，在对象生命期中不会改变。

- ✧ 对象成员：用户定制数据类型的数据成员。
 - ✧ 静态数据成员：**static** 修饰的数据成员，所有对象共享的数据成员。
 - ✧ 数据成员的初始化：构造函数。
- (2) 类的成员函数：约定对象的行为特征。
- ✧ 常量成员函数：**const** 修饰的成员函数，描述不修改数据成员的成员函数。
 - ✧ 静态成员函数：**static** 修饰的成员函数，所有对象共享的成员函数。
 - ✧ 构造函数：约定对象产生的行为。
 - ✧ 析构函数：约定对象消亡的行为。
 - ✧ 操作符重载函数：约定对象运算的行为。

2.1.4 对象产生和消亡的时机

(1) 局部域 + 自动变量/对象

在变量/对象的定义点产生（从栈区获得存储空间），在离开局部域时消亡。

(2) 局部域 + 静态变量/对象

在程序第一次执行到其定义点时产生（从静态数据区获得存储空间并进行初始化工作），此后持续存在，直到程序执行结束时消亡。

(3) 全局变量/对象

在程序开始执行时就产生（从静态存储区获得存储空间并进行初始化工作），此后持续存在，直到程序执行结束时消亡。

(4) 动态变量/对象

调用 **new** 运算符时产生（从堆区获得存储空间），调用 **delete** 运算符时消亡。

2.1.5 构造函数：约定对象产生的初始化行为

(1) 构造函数（**constructor**），是专门用于对象初始化的成员函数，由系统在对象产生时自动调用，并且在该对象从产生到消亡的一生中就只会调用这一次。

(2) 定义语法：

类名(形参列表) 《: 成员初始化列表》 { }

(3) 构造函数分类：

- ✧ 默认构造函数：无参数，没有提供任何初始状态信息；例如，

```
CDate now; //调用默认构造函数
```

- ✧ 普通有参构造函数：有参，提供了一些初始状态信息；例如，

```
CDate today(2008, 5, 1); //调用有参构造函数
```

- ✧ 拷贝构造函数：单参数，提供了另一个同类对象；例如，

```
CDate birthday = today; //调用拷贝构造函数，或者
CDate birthday(today);
```

- ✧ 转换构造函数：单参数，提供了另一个非同类对象；例如，

```
CDate tomorrow = "2005-11-5"; //调用转换构造函数，或者
CDate tomorrow("2005-11-5");
```

- (4) 定义默认构造函数。若不提供，系统合成一个默认构造函数。

```
CDate::CDate() { year=1900; month=1; day=1; }
//CDate::CDate() : year(1900), month(1), day(1) {}
```

- (5) 定义普通有参构造函数

```
CDate::CDate(int y, int m, int d):year(y),month(m),day(d){}
```

- (6) 定义拷贝构造函数

```
CName(CName &clone_me) { .....} //或者,
CName(const CName &clone_me) { .....} //或者,
CName(CName &clone_me, int n=10) { .....}
```

- (7) 默认拷贝构造函数：

- ✧ 不提供拷贝构造函数，编译器会合成一个默认的拷贝构造函数。
- ✧ 默认拷贝构造函数的行为：递归地为所有基类和对象数据成员调用拷贝构造函数，或者说按成员拷贝的方式进行初始化，等效于用源对象的每个数据成员来初始化目标对象的对应数据成员。

- (8) 何时调用拷贝构造函数

- ✧ 在需要通过同类对象克隆产生新对象的任何时候，调用拷贝构造函数。
- ✧ 常见情况：(a) 函数按值返回一个对象；(b) 函数调用时克隆生成形参对象。

- (9) 转换构造函数：在类型转换时调用。

```
CDate(string date); //转换构造函数
```

- (10) 显式转换构造函数：只有在显式类型转换时才会调用。


```
explicit CDate(string date); //转换构造函数（显式）
```

(11) 转换成员函数

```
operator 转换的目标类型名();
```

2.1.6 析构函数：约定对象死亡的善后行为

(1) 析构函数，专门用于约定对象消亡时的善后清理行为。

```
CDate::~~CDate(){……}
```

(2) 对象的深拷贝与浅拷贝：实现深拷贝需要提供 4 个特殊的成员函数：

- ✧ 构造函数：为指针分配存储空间，或者将其置为空（NULL）；
- ✧ 析构函数：释放动态分配的存储空间；
- ✧ 拷贝构造函数：拷贝动态分配的存储空间；
- ✧ 赋值操作符重载函数：拷贝动态分配的存储空间。

2.1.7 操作符重载函数：约定对象运算的行为

(1) 重载操作符，可以采用成员或者非成员重载方式，其定义语法形式分别为：

```
返回类型类名::operator 操作符(形参列表) { …… } //成员形式
```

```
返回类型 operator 操作符(形参列表) { …… } //非成员形式
```

(2) 对于操作符#，将其运算表达式转换为等价的直接函数调用形式，即可反向推演出操作符重载函数的声明原型。以二元操作符减法为例，

运算表达式：obj1 - obj2，

等价的成员函数调用形式：obj1.operator-(obj2)

等价的非成员函数调用形式：operator-(obj1, obj2)

(3) 各种操作符的运算表达式及其成员和非成员调用

表 1 操作符的成员和非成员调用形式

运算符	运算表达式	成员调用	非成员调用
二元	obj1 # obj2	obj1.operator#(obj2)	operator#(obj1,obj2)
一元前置	#obj	obj.operator#()	operator#(obj)
一元后置	obj#	obj.operator#(0)	Operator#(obj,0)
赋值=	obj1 = obj2	obj1.operator=(obj2);	不允许
下标[]	obj[i]	obj.operator[](i)	不允许
箭头->	obj->	obj.operator->()	不允许
函数调用()	obj(参数表)	obj.operator()(参数表)	不允许

(4) 如何选择成员还是非成员方式进行操作符重载

- ✧ 成员方式要求操作符的左操作数必须是该类的对象。
- ✧ 如果操作符的左操作数可能或者必须是其他数据类型时，我们就应该采用非成员的方式，将操作符重载为全局函数。
- ✧ 如果一个类已经定义完成，不希望再向其中添加任何成员函数，此时自然就必须选择非成员方式进行重载。
- ✧ C++语言规定，赋值操作符=、下标操作符[]、函数调用操作符()、指针访问操作符->，必须被定义为类的成员函数。

2.1.8 约定类成员的常量性

可以使用 `const` 关键字约定类的常量数据成员和常成员函数。其中，

(1) 常量数据成员：约定对象不变的属性，`const` 修饰；

- ✧ `const` 数据成员是对象的常量，在对象一生中保持状态不变，必须进行初始化，且只能在构造函数中通过初始化列表显式或者隐式地初始化。
- ✧ 引用型数据成员的初始化，与常量数据成员类似。

(2) 常成员函数：约定对象不修改状态的行为，`const` 修饰；

- ✧ `const` 是常成员函数类型描述的一部分，`const` 可用于重载区分。
- ✧ `const` 成员函数中不允许调用非 `const` 成员函数，只能调用其他 `const` 成员函数。
- ✧ `const` 对象只能访问 `const` 成员函数，非 `const` 对象则可以访问 `const` 或者非 `const` 成员函数。

2.1.9 约定类成员的静态性

使用 `static` 关键字约定静态数据成员和静态成员函数。其中，

(1) 静态数据成员：约定类域名字空间中的全局变量，所有对象共享 `static` 数据成员的唯一副本。

- ✧ 静态数据成员的声明和定义：在类体中只能进行静态数据成员的声明，必须在类体外的文件域中给出静态数据成员的定义和初始化。
- ✧ 静态数据成员的访问：

```
类名::静态数据成员名; //或者,
```

```
任意的对象名.静态数据成员名;
```

(2) 静态成员函数：约定类域名字空间中的全局函数。

✧ 静态成员函数的定义：`static` 修饰。

✧ 静态成员函数的访问：

```
类名::静态成员函数名(参数表); //或者,
```

```
任意对象名.静态成员函数名(参数表);
```

✧ 静态成员函数中不存在自身对象，其 `this` 指针无效。

✧ 在静态成员函数中只能访问静态成员，不能访问非静态成员。

2.2 对象的组合与继承

C++语言提供**组合**和**继承**两种工具支持基于组合和类属关系定制新的数据类型。

2.2.1 基于组合关系定制类：组合类

(1) 组合类的某个数据成员是另一种定制数据类型，称为对象成员或子对象。

(2) 组合类对象的构造和析构：

✧ 在产生时总是先调用对象成员的构造函数，然后再调用自身的构造函数。

✧ 消亡时先调用自身的析构函数，然后再析构对象成员。

✧ 对象成员之间构造和析构的顺序，按照它们在类体中声明的顺序进行。

2.2.2 基于类属关系定制类：继承与派生

(1) 语法

```
class 派生类名 : 继承方式基类名 { //派生类继承基类成员
    //在类体中改造基类成员，或者说明派生类特有的成员
};
```

(2) 通过继承定制派生类，通常需要经过如下的四个步骤：

✧ 继承基类成员：确定继承的基类，选择从基类继承的方式；

✧ 改造基类成员：修改基类成员的访问限定或者重定义基类成员；

✧ 发展派生类成员：定义派生类自身特有的数据成员和成员函数；

- ✧ 重写派生类的构造和析构函数，如果有必要，重写赋值操作符。

(3) 选择继承方式

- ✧ 公有继承，基类的公有和保护成员被继承为派生类的公有和保护成员。

基类的私有成员→派生类的不可访问成员；

基类的公有成员→派生类的公有成员；

基类的保护成员→派生类的保护成员。

- ✧ 私有继承，基类的公有和保护成员被继承为派生类的私有成员。

基类的私有成员→派生类的不可访问成员；

基类的公有成员→派生类的私有成员；

基类的保护成员→派生类的私有成员。

- ✧ 保护继承，基类的公有和保护成员被继承为派生类的保护成员。

基类的私有成员→派生类的不可访问成员；

基类的公有成员→派生类的保护成员；

基类的保护成员→派生类的保护成员。

(4) 改造基类成员

- ✧ 修改基类成员的访问限定（`using`）；

在派生类中可以使用 `using` 访问声明改变来自基类的成员的访问限定，但是，基类私有成员是派生类的不可访问成员，不能修改其访问限定。

- ✧ 修改基类成员的定义（同名隐藏）。

在派生类中可以根据需要重新定义来自基类的成员。此时，重新定义的基类成员，会隐藏原来同名的基类成员。必须注意，隐藏不是重载。

(5) 派生类对象用作基类对象的赋值兼容

在**公有继承**的情况下，派生类对象可以被当作基类对象来使用，反过来则是禁止的。具体的赋值兼容规则表现为：

- ✧ 派生类对象，可以用来初始化或者赋予基类对象；
- ✧ 派生类对象，可以用来初始化基类引用；
- ✧ 派生类对象的地址，可以用来初始化或者赋予基类指针。

(6) 基类到派生类的动态类型转换

通过 `dynamic_cast` 在运行期间进行动态类型转换。此时，`dynamic_cast` 会在运行期间进行类型检查，如果引用或指针所指向的对象不是转换的目标类型对象，则 `dynamic_cast` 会失败，且给出相应的错误指示，

- ✧ `dynamic_cast` 到指针，则转换失败的结果指针为 `0` 值；
- ✧ `dynamic_cast` 到引用，则转换失败会抛出一个 `bad_cast` 异常。

(7) 多态：根据对象类型不同调用不同函数的行为。

- ✧ 函数重载：编译器多态
- ✧ 虚函数：运行期多态

(8) 虚函数

- ✧ 定义：`virtual` 修饰的成员函数。
- ✧ 覆盖：在派生类中重新定义基类虚函数。
- ✧ 引发多态的条件：当且仅当通过基类引用或者基类指针调用虚函数的时候，系统就会根据实际对象类型，在运行期间自动地选择合适的虚函数版本，实现多态行为：
 - (a) 如果基类引用/指针指向基类对象，则绑定虚函数的基类版本；
 - (b) 如果基类引用/指针指向派生类对象，则绑定虚函数的派生类版本；
- ✧ 覆盖、重载、同名隐藏
- ✧ 虚函数的运行期绑定原理：`virtual` 修饰导致编译器(a)创建虚函数表；(b)添加指向虚函数表的指针成员；(c)改写虚函数调用代码。
- ✧ 虚析构函数：如果需要通过基类指针释放派生类对象，将析构函数声明为虚函数。如果类中不包含虚函数，则不要把析构函数声明为虚函数。

(9) 纯虚函数和抽象类

- ✧ 具体类：可以产生对象的类。
- ✧ 抽象类：包含纯虚函数的类。抽象类不允许产生对象，可以定义抽象类的指针和引用。抽象类的派生类，必须为抽象基类的全部纯虚函数提供定义，否则该派生类依然是抽象性。
- ✧ 纯虚函数：具有初始式“`=0`”的虚函数。
- ✧ 接口类：没有数据成员，只有纯虚成员函数。

2.3 异常处理

(1) 异常检测和异常处理的分离

当函数代码可能发生某种异常且暂时无法处理这种异常时，可以抛出这种异常（即**抛出异常**）；然后，函数调用者就可以检测是否真正发生了异常（即**捕获异常**），并且说明若发生了这种异常就进行什么样的补救处理（即**处理异常**）。

(2) C++语言的异常处理机制就提供了如下的语法工具和设施，包括，

- ✧ **throw** 语句：用于函数抛出特定异常，通知异常发生；
- ✧ **try-catch** 语句：**try** 用于捕获异常，**catch** 用以处理异常。

3. 泛型思维与泛型编程

泛型问题求解思维，认为通过将特殊问题中的某些因素设为可变参数，可以将特殊问题泛化为适用范围更广的一般问题，此后就可以借助于一般问题的求解思路来完成特殊问题的求解。泛型编程，就是在泛型思维下编写程序描述问题求解思路，需要在数据或者对象参数化的情况下描述和应用问题求解思路，包括：

(1) 泛化环节：基于泛化的数据或者对象类型来描述通用的问题求解思路；

(2) 应用环节：设定通用求解思路的泛化参数，描述特定具体问题的求解过程。

C++语言提供了模板机制来支持数据类型的参数泛化，包括处理过程泛化的函数模板以及处理对象泛化的类模板。另外，C++语言基于模板实现了常用的一些算法和数据结构，形成了一个具有工业级强度、健壮可靠的泛型程序库，称为标准模板库（**Standard Template Libaray, STL**），是我们进行泛型编程的基础。

3.1 泛化数据类型：定义模板

C++语言的模板语法工具支持函数和类的泛化，即函数模板和类模板。

3.1.1 函数模板

(1) 定义函数模板

```
template<模板参数表>
返回类型函数名(形式参数表) { ..... } //函数体
```

(2) 模板参数有两种：类型参数和非类型参数。其中，

✧ 类型参数：表示该参数是未知数据类型或对象类型，说明语法：

```
typename 类型名 //或者：class 类型名
```

✧ 非类型参数：表示该参数不是类型，而是未来要指定的常量值。

(3) 例如：

```
template<typename T>
T max(const T arr[] , int size){ }
```

3.1.2 类模板

(1) 定义类模板

```
template<模板参数表>
class 类名 { ..... } //类体
```

(2) 例如：

```
template<class T> //T 是类型参数
class Array { ..... } //类体
```

3.2 应用环节：模板实例化

函数模板不是函数，类模板也不是类，不能直接使用；必须将函数模板和类模板中的模板参数替换成具体类型或者常量值，才能产生真正的函数代码和类代码，这个步骤称为**模板实例化**。

(1) 模板实例化方法：隐式实例化（在需要时，编译器根据模板定义生成模板实例）、显式实例化（程序员直接声明要求模板即刻完成实例化）。

(2) 模板参数推理：显式指定、隐式推理

(3) 模板特化：在特定或者限定模板参数的情况下为模板提供特殊的定义，包括完全特化、部分特化。函数模板支持完全特化，类模板支持完全和部分特化。

3.3 C++标准模板库

C++语言利用模板机制，实现了计算机科学领域常用的算法和通用数据结构，形成具有工业级强度、健壮可靠的函数模板和类模板集合，称为标准模板库（STL）。

3.3.1 标准模板库的功能

C++语言的标准模板库可以划分为六种类型的功能组件：容器（**container**）、迭代器（**iterator**）、算法（**algorithm**）、仿函数（**function object**）、适配器（**adapter**）、分配器（**allocator**）。利用这些功能组件，我们可以：

（1）把需要处理的数据或者对象放到**容器**中，进行有效的组织和存储。容器是实现各种数据结构的类模板集合，如 **vector**、**list**、**map** 等。

（2）利用**迭代器**对**容器**中的元素进行存取访问。C++语言的迭代器是访问容器的泛型指针类模板，每种容器均附带自身的迭代器。

（3）通过迭代器指定容器区间范围，利用**算法**对该区间范围内的容器元素进行操作和处理。算法是一组常用函数模板，如 **sort**、**find**、**copy** 等。

（4）仿函数，是重载了 **operator()** 的类或类模板，其行为类似于函数，使用**仿函数**可以对容器元素的操作和处理行为进行定制。

（5）适配器，用于对容器、迭代器、仿函数的行为进行转换。

（6）分配器，用于分配和管理容器的存储空间，标准模板库的容器均使用一种默认的通用分配器。

3.3.2 标准模板库的容器

标准模板库的容器，用于容纳和管理其他的对象，每种容器都附带了自身的迭代器用来访问和操作容器中所包含的元素。这些容器总体上分两类七种：

（1）顺序式容器：向量 **vector**、列表 **list**、双向队列 **deque**。

（2）关联式容器：集合 **set**、多重集合 **multiset**、映射 **map**、多重映射 **multimap**。

表 2 顺序容器 **vector**、**list**、**deque** 的特性

特征&操作	vector	list	deque
内部结构	动态数组	双向链表	多个数组
随机访问	支持，很快	不支持	支持，比 vector 慢
顺序访问	支持	支持	支持
查找速度	慢	很慢	慢
插入删除元素	支持，在尾部很快，其他位置很慢	支持很快	支持，首尾位置很快，其他位置较慢

表 3 关联容器 `set`、`map`、`multiset`、`multimap` 的特性

特征&操作	<code>set</code>	<code>multiset</code>	<code>map</code>	<code>multimap</code>
特征	键集合，键不可重复	允许重复键的 <code>set</code>	键-值对集合，键不允许重复	允许重复键的 <code>map</code>
内部结构	二叉搜索树	二叉搜索树	二叉搜索树	二叉搜索树 ¹
随机访问	不支持	不支持	按 <code>key</code> 随机访问	不支持
查找速度	快	快	快	快
插入删除元素	支持	支持	支持	支持

(3) 顺序容器的适配器：栈 `stack`、队列 `queue`、优先队列 `priority_queue`。

- ✧ 栈：将容器元素的插入和访问行为约束转换为后进先出顺序。
- ✧ 队列：将容器元素的插入和访问行为约束为元素先进先出顺序。
- ✧ 优先队列：将容器元素的插入和访问行为约束为按优先级度量顺序。

3.3.3 迭代器

迭代器，是指针概念的泛化和推广，通过迭代器，可以像指向数组的指针那样访问到容器中的每个位置，进而访问该位置的元素。

(1) 五种迭代器：输入迭代器（单步向前迭代 + 读取数据）、输出迭代器（单步前向+单次写入数据）、前向迭代器（输入输出迭代器+多次读写）、双向迭代器（输入输出迭代器+双向单步）、随机访问迭代器（双向迭代器+任意跳转）。

(2) 容器附带的迭代器：

- ✧ 标准模板库的每种容器类型都附带了自己的迭代器。
- ✧ `vector` 向量容器提供了三种类型的迭代器：随机访问迭代器、逆向随机访问迭代器、常迭代器。

```
vector<int>::iterator //随机访问迭代器，可读写
vector<int>::reverse_iterator//逆向随机访问迭代器，可读写
vector<int>::const_iterator//随机访问迭代器，可读不可写
```

(3) 迭代器的适配器

- ✧ 通过迭代器适配器，可以为迭代器提供更多的操作功能。

¹二叉搜索树是一种有序的二叉树数据结构，左子树节点值都小于父节点，右子树节点值都大于根节点，插入、查找的效率都很高，具体细节请参阅有关数据结构的书籍或网站。

- ✧ 三种常用的迭代器适配器：逆向、流和插入适配器。
- ✧ 三种类型的插入迭代器：尾插入迭代器（`back_inserter`）、头插入迭代器（`front_inserter`）、一般插入迭代器（`inserter`）。

3.3.4 算法

（1）查找算法：用各种策略去判断容器中是否存在指定元素，包括在无序容器中查找（`find`、`count`）、在有序容器中查找、序列匹配三种类型。

（2）排序和整序算法：`sort`、`stable_sort`、`merge`、`inplace_merge`、`nth_element`、`random_shuffle`、`rotate`、`rotate_copy`。

（3）拷贝、删除和代替算法：`copy`、`copy_backward`、`iter_swap`、`swap`、`swap_range`、`remove`、`remove_copy`、`remove_if`、`remove_copy_if`、`replace`、`replace_copy`、`replace_if`、`replace_copy_if`、`unique`、`unique_copy`。

（4）排列组合算法：`next_permutation`（按字典序计算给定排列的下一个排列）、`prev_permutation`（按字典序计算给定排列的上一个排列）。

（5）生成和改变算法：`generate` & `generate_n`、`fill` & `fill_n`、`for_each`、`transform`。

（6）关系比较算法：`equal`、`includes`、`lexicographical_compare`、`max` & `max_element`、`min` & `min_element`、`mismatch` 等。

（7）集合算法：`set_union`、`set_intersection`、`set_difference` 等。

（8）算术算法：`accumulate`、`partial_sum`、`inner_product` 等。

（9）堆算法：`make_heap`、`push_heap`、`pop_heap`、`sort_heap` 等。

疯狂实践系列一：素数判定问题

题目：从键盘输入一个正整数 N ，判定它是不是素数，输出判定结果。

1. 训练目标

- (1) 准备编程环境，学习如何新建工程和源代码文件。
- (2) 编写什么都不做的 C++ 程序，掌握程序的固有结构。
- (3) 学习如何利用注释描述素数判定问题的解题思路。
- (4) 学习如何进行程序的编译和链接，如何执行程序。
- (5) 如何声明和定义函数；如何描述整型数据和 `bool` 型数据；如何处理数据的输入和输出；如何对数据进行运算处理；如何控制数据的加工处理流程。

2. 准备编程环境

疯狂实践系列课程的编程环境，采用 Visual Studio 2010（简称 VS2010）。其他开发工具类同，请读者自行探索使用方法。图 2 是 VS2010 打开的初始界面。

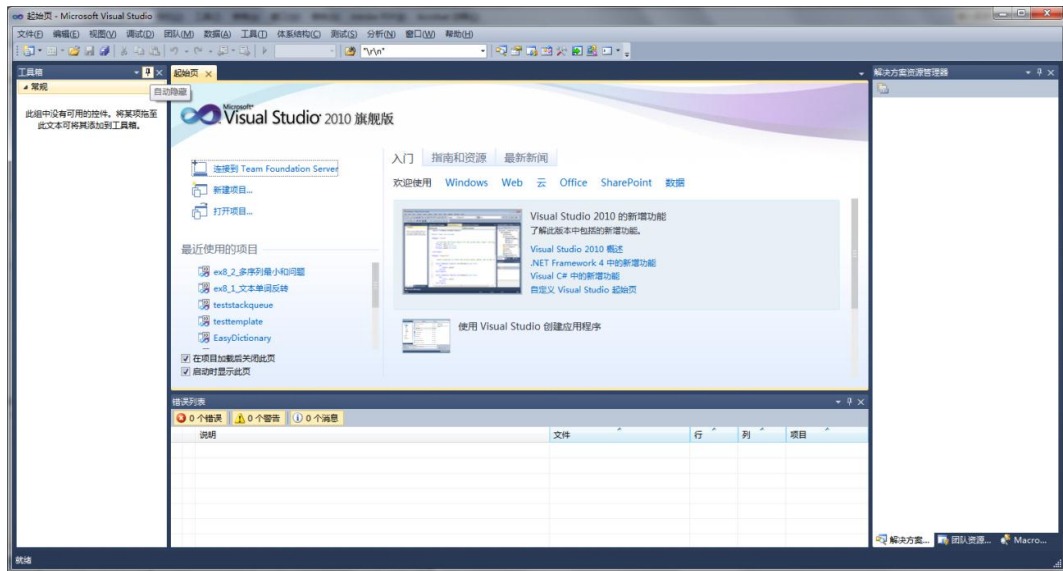


图 2 Visual Studio 2010 集成开发环境的初始界面

2.1 新建一个工程项目 (Project)

打开 Visual C++ 集成开发环境。选择【文件】File 菜单，点击【新建/项目】，出现图 3 所示的界面。

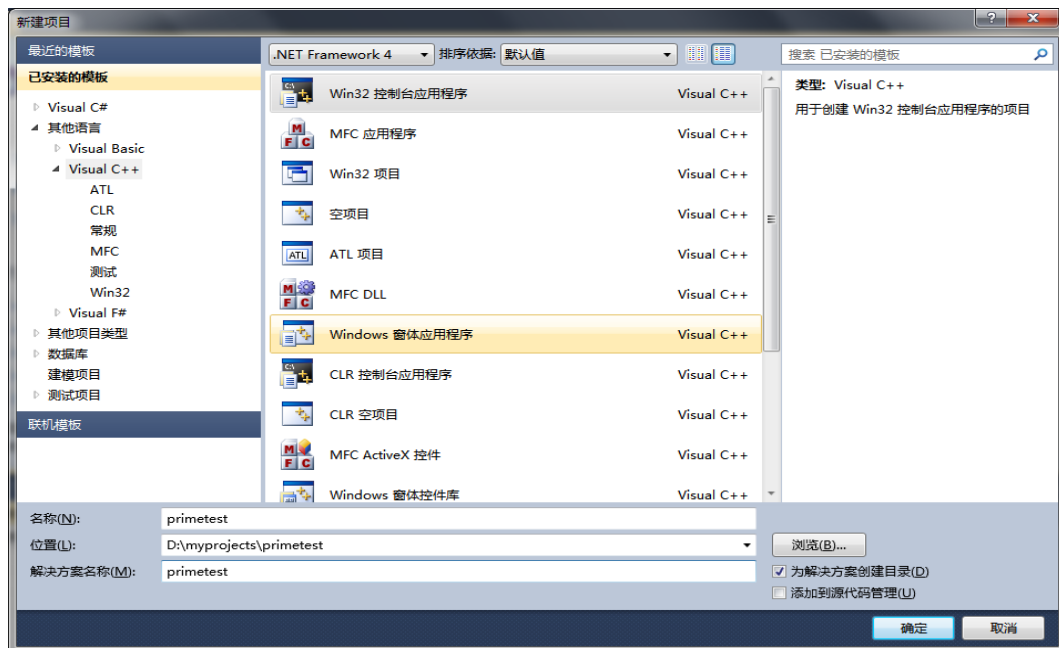


图 3 新建项目向导：项目信息设置界面

在左侧窗格中选择“Visual C++”，中间窗格中选择“Win32 控制台应用程序”，右侧窗格中就显示了当前选择的项目类型。

在下方“名称 (N)”文本框中，输入项目名称，如 `primetest`；然后，选择该项目打算保存在什么位置，假设是：`D:\myprojects\primetest`。

点击【确定】按钮。此时，会出现“Win32 应用程序向导 - `primetest`”界面，直接点击界面下方的【完成】按钮。此时，就生成了 `primetest` 项目，以及该项目所隶属的解决方案 `primetest (solution)`，具体界面如图 4 所示。

在图 4 中，左侧窗格是“解决方案资源管理器”，其下管理着隶属该解决方案的所有项目，此时解决方案 `primetest` 下只有一个隶属的项目，项目名称为 `primetest`；在项目 `primetest` 下管理着一系列的文件，包括头文件、源文件、资源文件、外部依赖项等。其中，源文件有两个：`primetest.cpp` 和 `stdafx.cpp`。

暂时不要理会名字很怪异的那个文件 `stdafx.cpp`，现在，我们只要知道接下来编写程序的工作就在源文件 `primetest.cpp` 中进行即可。

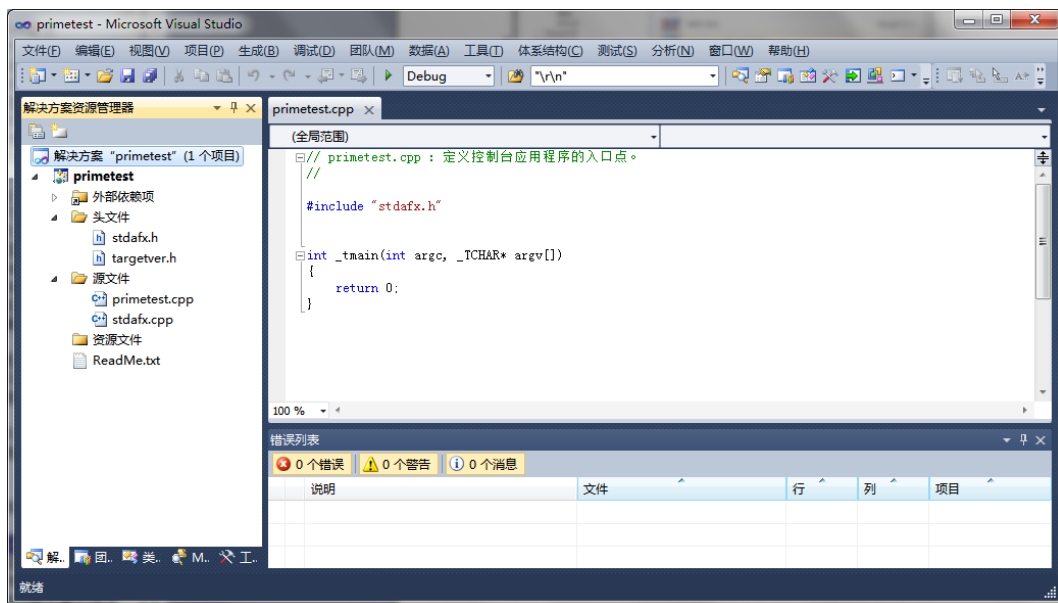


图 4 新建项目向导：新建项目完成的界面

2.2 C++程序的固有结构

创建了工程和源代码文件之后，接下来就要在源代码文件 `primetest.cpp` 中进行程序代码的编辑，即编写程序来描述素数判定问题的解题思路。

不过，在具体编程求解素数判定问题之前，首先让我们在源文件 `primetest.cpp` 中，输入如下代码，编写一个语法完整、但是什么都不做的 C++ 程序。

```
//primetest.cpp
#include "stdafx.h"
#include <iostream>
using namespace std;
int main( ){
    return 0;    //返回整数 0，表示正常返回操作系统
}
```

图 5 一个语法完整最简单的 C++ 程序

注意，在源代码文件 `primetest.cpp` 中，通常已经由向导工具生成了类似的代码。建议把向导生成的程序代码删掉，照着上面的代码老老实实的敲一遍。

在源文件 `primetest.cpp` 中，重点关注那个名字为 `main` 的函数。一个语法完整的程序至少应包含一个开始执行的入口函数——`main` 函数，作为程序执行的起点。显然，对于一个程序来讲，不能有多个执行的起点，即有且只有一个 `main` 函数。

C++标准约定，`main` 函数应该具有如下的样子：

```
int main( ) {  
    ..... //在此处添加你的程序代码  
    return 0;  
}
```

其中，`main` 是函数的名字，`main` 之后的括号 `()` 中放置**形式参数列表**，括号中为空表示没有形式参数。`main` 之前的 `int` 表示函数的**返回值类型**，C++标准约定，`main` 函数必须返回一个整数类型的返回值。函数名 `main`、参数列表、返回类型构成函数头。

函数头之后，由花括号 `{}` 括起来的是函数体。函数实现代码就书写在函数体中。`main` 函数要求一个整型返回值，因此，在函数体最后一行用到 `return` 语句：

```
return 0; /* 根据要求，返回一个整数 0 */
```

在 C++语言中，`return` 是一种跳转语句，用于将程序执行流程返回给函数调用者，并传回指定的返回值。`main` 函数的调用者是操作系统，所以，这里的 `return` 语句会将执行流程返回给操作系统，并传回指定返回值 `0`，然后程序执行结束。通常约定，`main` 函数返回 `0` 表示程序正常退出，非 `0` 表示程序异常退出。

注意 `return` 语句的最后有一个分号 `;`。C++语言中，用分号表示一条语句结束了。

上述代码是 C++程序的固定结构。在编写程序时，可以首先输入这个语法完整的 C++程序，然后，再在其中添加与问题求解相关的程序代码。

在 `main` 函数之前，程序的第一行代码，是一条用于包含头文件的预编译指令，

```
#include <iostream>
```

`#include` 称为包含指令，尖括号 `<>` 中的 `iostream` 是一个文件名字，称为 C++头文件。整个指令是将 `iostream` 这个头文件包含进我们的源程序中，或者说，将 `iostream` 文件中的程序代码，拷贝到该指令所在位置。`iostream` 文件是 C++语言预先定义好的头文件，其中的程序代码可以帮助我们完成数据的键盘输入和屏幕输出显示。

由于 `iostream` 文件中的名字都隐藏在一个叫做 `std` 的名字空间中，因此，需要将 `std` 名字空间曝光于天下，让我们可以看到其中的标识符，即使用如下指令：

```
using namespace std;
```

现在暂不深究预编译指令、头文件、名字空间这些术语的含义和用法，只要知道图 5 就是一个语法完整的 C++ 程序的固有结构，记住它即可。

3. 编写素数判定程序

语法完整但什么都不做的 C++ 程序自然没什么用处，我们需要编写程序代码，描述素数判定问题的解题思路。在一种称为**过程化的思维**下面，求解问题就是对数据进行加工处理，使得数据状态发生一系列改变，最终达到我们期望的目标状态的过程。

所以，在过程化问题求解思维下，对于任意给定问题我们都需要思考：（1）求解这个问题需要准备哪些数据；（2）对这些数据进行哪些加工处理可以得到期望的结果。

相应地，在编写程序代码时需要描述四个环节的任务：（1）准备数据；（2）输入数据；（3）加工处理数据；（4）输出处理结果。

3.1 利用注释描述问题求解的框架

现在先不急于编写程序代码（着急也没用），让我们利用 C++ 语言的注释文字把问题求解思路的框架先描述下来，形成图 6 所示的问题求解框架。

```
//primetest.cpp
#include "stdafx.h"
#include <iostream>
using namespace std;
int main( ) {
    //1. 描述待处理的数据
    //2. 输入待处理的数据
    //3. 处理数据：判定是否素数
    //4. 输出处理结果
    return 0;    //返回整数 0，表示正常返回
}
```

图 6 素数判定的问题求解思路框架

注释，是对程序代码的文字解释说明，用于帮助程序员阅读程序。注释文字会在程序编译时被忽略掉，不影响程序执行。C++ 语言提供了两种注释方式：

（1）行注释：“//”表示注释开始，其后是注释文字，换行表示注释结束；

(2) 块注释: “/*”表示注释开始, “*/”表示注释结束, 其间出现的文字都是注释。块注释可以容纳多行的注释文字。

在 `primetest.cpp` 的 `main` 函数中使用了行注释方式。根据注释文字可以看出: 素数判定问题的过程化求解思路, 需要完成如下四个任务才能实现问题求解: (1) 描述待处理的数据; (2) 输入待处理的数据; (3) 处理数据: 判定是否素数; (4) 输出处理结果。

3.2 把注释翻译为 C++ 代码

注释文字描述了问题求解的框架, 我们需要把注释文字翻译成 C++ 程序代码。

(1) 翻译 “1-描述待处理的数据”

利用 C++ 的变量和数据类型, 定义一个整型变量 `iNum`, 用来容纳待判定的整数:

```
int iNum; //通过C++的变量和数据类型来描述数据
```

(2) 翻译 “2-输入待处理的数据”

利用 C++ 标准输入流对象 `cin` 和输入运算符 “>>”, 可以将整数的具体取值从键盘输入到整数变量 `iNum` 中, 完成输入数据的工作:

```
cin>>iNum; //通过C++标准输入流来输入数据
```

(3) 翻译 “3-处理数据: 判定是否素数”

第 3 个任务 “处理数据: 判定是否素数” 比较复杂, 难以一步翻译到位。不过, 我们可以假设存在一个机器能判定一个正整数是不是素数, 机器的名字叫做 `isprime`, 如果把一个正整数扔到机器中, 则机器就能判断出这个整数是不是素数, 是素数则输出 `true`, 不是则输出 `false`。在 C++ 语言中, 描述这种数据处理机器的语法工具就是 **函数**。

我们假设存在一个名字叫做 `isprime` 的函数, 函数大概应该具有如下的样子:

- a) 函数的名字: `isprime`。
- b) 函数的入口参数: 一个 `int` 整型变量。
- c) 函数的返回值类型: `bool` 型, `true` 表示是素数, `false` 表示非素数。
- d) 函数的功能: 判定输入的整型变量是否素数。

根据上面的设想, 总结一下我们需要这样一个函数: `bool isprime(int)`。如果存在这样的机器 (函数), 则使用该机器就可以判定一个正整数是否素数, 术语称为 **调用函数**。具体做法是把待正整数 `iNum` 放到该函数入口中, 代码如下:

```
isprime(iNum)
```


然后函数就运转起来判断 `iNum` 是不是素数，最后返回判定结果：`true` 或 `false`。其中 `true` 表示放进了的 `iNum` 是素数，`false` 表示 `iNum` 不是素数。

可以把返回的判定结果保存起来，放到一个名字叫做 `bFlag` 的 `bool` 型变量中，`bool` 型变量是专门用于保存 `true` 或 `false` 数据的容器。

因此，对于第 3 个任务：判定是否素数，可以将其程序代码翻译为：

```
bool bFlag = isprime(iNum);
```

(4) 翻译“4-输出处理结果”

`bFlag` 中保存着素数判定结果，所以第 4 个任务“输出处理结果”就很容易处理。

根据 `bFlag` 的取值情况，在屏幕上输出判定结果。如果 `bFlag` 是 `true`，说明 `iNum` 是素数，则在屏幕上输出 `iNum` “是素数”；否则输出 `iNum` “不是素数”。

C++语言中，“如果…就…否则…”的逻辑，可以用 `if-else` 语句描述；向屏幕输出文字，可以通过标准输出流对象 `cout` 和输出运算符“<<”来完成：

```
if(bFlag)
    cout<<iNum<<"是素数"<<endl;
else
    cout<<iNum<<"不是素数"<<endl;
```

至此，注释文字的四个任务初步翻译完成，得到如下的源代码：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main() {
    //1. 描述待处理的数据
    int iNum; //通过 C++的变量和数据类型来描述数据
    //2. 输入待处理的数据
    cin>>iNum; //通过 C++标准输入流来输入数据
    //3. 处理数据：判定是否素数
    bool bFlag= isprime(iNum); //通过函数 isprime 判定 iNum 是否素数
    //4. 输出处理结果
    if(bFlag) //如果 bFlag 是 true, 则 iNum 是素数
        cout<<iNum<<"是素数"<<endl;
```

```
else//否则, iNum 不是素数
    cout<<iNum<<"不是素数"<<endl;
return 0;
}
```

3.3 尝试程序编译链接

源代码初步编辑完成后, 可以尝试进行程序的**编译和链接**工作。如果程序编写无误, 则编译会将源代码文件 (*.cpp) 翻译成机器语言形式的目标代码文件 (*.obj), 而链接则会产生最终的可执行文件。执行可执行文件, 就可以得到问题求解的结果。

3.3.1 编译源代码文件生成目标代码

选择菜单【生成/生成解决方案】, 或者使用快捷键 **F6**, 进行源代码的编译链接工作。编译结果提示有一个错误。错误信息如下:

错误信息说明	文件	行	列
error C3861: "isprime" : 找不到标识符	primetest.cpp	14	1

大概意思是说, 在 `primetest.cpp` 源文件中的第 14 行中出现了错误, 错误编号 C3861, 错误的原因是: 找不到 `isprime` 标识符, 编译器不认识它是什么东西。

我们在前面埋了一个定时炸弹: 假设存在一个数据处理的机器——`isprime` 函数。但是 `isprime` 函数实际并不存在, 故代码编译出错, 提示找不到标识符。

C++语言中, 可以采用一种暂时的解决办法: **函数声明**。在第 3 步调用 `isprime` 函数之前先声明一下 `isprime` 标识符。语法非常简单:

```
bool isprime(int);    //isprime 函数的原型声明
```

通过函数声明可以给编译器提供一种保证: “`isprime` 是一个函数, 在未来某个地方我一定会提供其具体定义, 你先放心大胆地编译吧!”。此时的程序代码如下:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
bool isprime(int);    //isprime 函数的声明
int main() {
    .....
```

```
//3. 处理数据：判定是否素数

bool bFlag = isprime(iNum); //调用函数 isprime 判定 iNum 是否素数
.....

}
```

3.3.2 链接目标代码生成可执行文件

再次选择菜单【生成/生成解决方案】，或者使用快捷键 F6，进行源代码的编译和链接。结果是编译成功，产生了目标代码文件：primetest.obj。

但是，链接失败，出现了两个链接错误。摘录其中一个错误信息如下：

错误信息说明	文件
error LNK2019:无法解析的外部符号"bool _cdecl isprime(int)" (?isprime@@YA_NH@Z)，该符号在函数 _main 中被引用	primetest.obj

大概意思是说，在目标代码 primetest.obj 文件中无法解析外部符号：isprime。即程序编译虽然没有错误，但在链接时却找不到 isprime 函数。确实是这样，我们只是声明存在这个函数，给编译器提供一种保证：未来一定会给出该函数的具体定义。但是，到目前为止，并没有给出函数定义。因此，我们需要提供 isprime 函数的定义。

3.4 添加 isprime 函数的定义

我们可以把 isprime 函数的定义跟 main 函数放在一起，都定义在 primetest.cpp 源代码文件中，当然也可以把它放到另一个的源文件中。

3.4.1 添加一个源文件

假设，我们想把 isprime 函数的定义放到一个名为 functions.cpp 的源代码文件中。那么，需要在项目中添加一个源文件，具体操作步骤如下：

(1) 右键选择项目“primetest”，在菜单中选择【添加/新建项】，如图 7 所示。

(2) 在跳出的“添加新项- primetest”界面中（图 8），选择“C++文件（.cpp）”，在名称文本框中输入源文件的名称，点击【添加】按钮。此时，就会在项目中添加一个新的源代码文件，文件名是 functions.cpp，并且打开该源文件作为当前编辑目标。

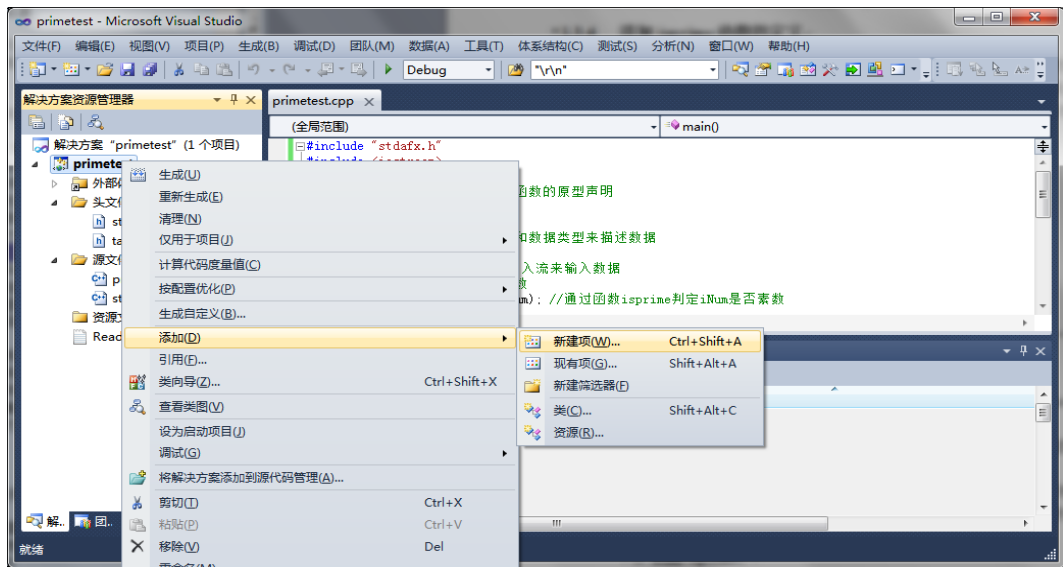


图 7 添加新建项的操作界面

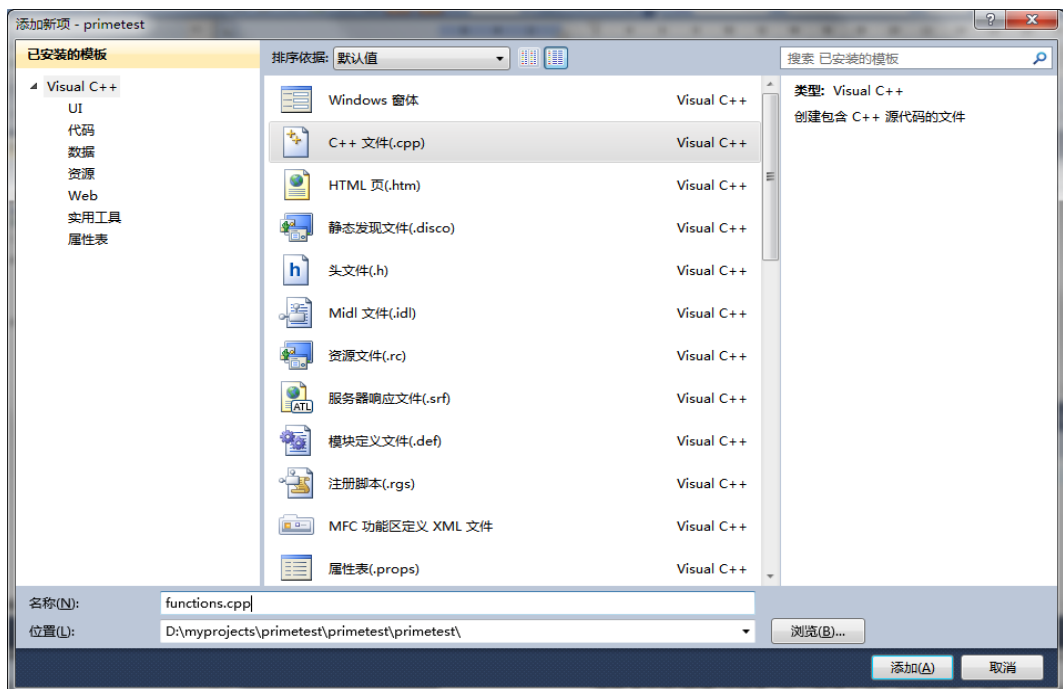


图 8 添加新建项向导：设置源文件信息

3.4.2 编写 isprime 函数代码

按照上述的操作步骤和方法，新建一个源文件 `functions.cpp`，然后在该源文件中编写程序代码，定义 `isprime` 函数的功能。

已知 `isprime` 函数的样子（术语称为**原型**）：`bool isprime(int)`，其功能是完成一个正整数是否是素数的判定，如是素数则返回 `true`，否则返回 `false`。

为了提高代码的可读性，还是先用块注释方法，描述 `isprime` 函数的相关规格信息，并编写出 `isprime` 函数定义的完整语法外壳：函数头和函数体。其中，函数体跟在函数头后面，用花括号`{}`括起来。具体的程序代码如下：

```
/*  
函数: isprime:  
功能: 判定整型变量 num 是否为素数;  
参数: num, 表示待判定的整数;  
返回: bool, true 表示是素数, false 表示不是素数。  
*/  
bool isprime (int num) //函数头  
{  
    //函数体, 在这里添加代码判定是否素数。  
}
```

接下来的工作，就是在花括号界定的函数体中描述 `isprime` 函数的功能，描述判定输入参数 `num` 是否素数的过程。同样，磨刀不误砍柴工，不要急于编写代码，先利用注释文字说明素数判定的思路。这里我们采用一种最简单的判定思路，得到如图 9 所示的函数代码，主要是注释文字。

```
//functions.cpp
```

```
/*  
函数: isprime:  
功能: 判定整型变量 num 是否为素数;  
参数: num, 表示待判定的整数;  
返回: bool, true 表示是素数, false 表示不是素数。  
*/  
bool isprime (int num) {  
    //任务 1: 用从 2 到 num-1 的一系列整数, 与 num 进行取余运算,
```

```

//任务 2: 如果遇到一个能够整除的, 则 num 必不为素数, 返回 false,
//任务 3: 如果没有遇到一个能够整除的, 则 num 必为素数, 返回 true。
}

```

图 9 函数 isprime 用注释描述的问题求解思路

然后继续把注释文字逐步翻译成 C++ 代码。当然, 如果注释文字所描述的任务难以一步翻译到位, 还可以继续假定存在另一个数据处理机器——函数, 能够完成注释文字所描述的工作, 然后调用函数、做函数声明进而定义函数的具体实现。

这里注释文字的翻译工作都比较容易。对于任务 1, 需要对 2 到 num-1 的每个整数反复与 num 进行取余运算, 可以借助 C++ 语言的 for 循环语句:

```

for(int i=2; i<num; i++) { //变量 i 从 2 开始, 每次加 1 直到 num-1,
    num % i; //num 和 i 进行取余运算, %表示取余运算
}

```

对于任务 2 和任务 3, 如果遇到一个能够整除的, 即某一次取余运算的结果等于 0, 那么 num 必不是素数, 则返回 false。显然, 这里需要表达一种“如果…就…”的选择逻辑, 可以借助 C++ 语言的 if 语句来实现, 返回用 return 语句实现:

```

for(int i=2; i<num; i++) {
    //如果存在取余运算的结果等于 0, 就结束运算返回 false
    if (0 == num % i) return false;    //==用于判断是否相等
}
//如果不存在取余运算结果等于 0 的情况, 就返回 true
return true;

```

在 for 循环过程中, 如果没有任何一次取余运算的结果是 0, 即不会 return false, 那么 num 必为素数, 则应该 return true。注意, 遇到 return 语句, 函数就会结束执行, 返回到函数被调用的位置, 同时传回返回值: true 或者 false。

至此, 注释文字的三个任务翻译完成, 实现了函数定义, 得到图 10 的源代码。

```

//functions.cpp

```

```

/*

```

```

函数: isprime:

```

```

功能: 判定整型变量 num 是否为素数;

```

```

参数: num, 表示待判定的整数;

```

```

返回: bool, true 表示是素数, false 表示不是素数。

```

```
*/  
bool isprime (int num) {  
    for(int i=2; i<num; i++) {  
        if (0 == num % i) return false;  
    }  
    return true;  
}
```

图 10 函数 isprime 翻译完成的程序代码

3.5 程序编译链接和执行

完成源代码文件的编辑之后，选择菜单【生成/生成解决方案】，或者使用快捷键 F6，进行源代码的编译链接工作。编译和链接均成功：

0 个错误 0 个警告 0 个消息

此时，编译器会成功生成两个目标代码文件 `primetest.obj` 和 `functions.obj`，在 `functions.obj` 中已经给出了 `isprime` 函数的定义代码，满足了链接处理的要求。编译器继续进行目标代码的链接工作，生成最终的可执行文件 `primetest.exe`。

此时，倘若打开资源浏览器，在工程所在目录下，可以找到一个 `Debug` 目录，包括了程序编译和链接所产生的一系列文件，其中：

`functions.obj`——编译 `functions.cpp` 源代码产生的目标文件
`primetest.obj`——编译 `primetest.cpp` 源代码产生的目标文件
`primetest.exe`——链接产生的可执行文件

有了可执行文件 `primetest.exe`，就可以执行我们的素数判定程序，选择【调试/启动调试】或者按 F5 键执行程序。在黑色的控制台界面中输入整数 121，然后回车，执行结果如下图所示，我们可以尝试多次执行程序输入其他正整数。

```
121          (键盘输入 121，然后按回车键)  
121 不是素数
```

至此，素数判定问题的编程求解大功告成！整个求解过程，始于创建项目和源代码文件，然后是编辑源代码，编译、链接、执行四个环节。其中，编辑源代码工作由我们程序员来完成，编译、链接和执行程序则由计算机的编译器来完成。

剔除代码中反映我们思维过程的注释文字，一个表面上完美无瑕的程序就此诞生（如图 11 所示），但是在完美形式背后漫长甚至有点丑陋的思维分析和迭代过程却消失不见，而这些不漂亮的思维分析和迭代过程才是学习者应该体会和关注的核心。

通过素数判定程序的编写实践，我们可能会体验到一种自顶向下、分而治之、逐步求精的编程方式，即将较大的任务逐步分解为一系列规模更小的子任务，直至子任务可以轻而易举地编程实现，专业术语称为“结构化编程思想”。

在过程化编程实践中，不要奢望一次就编写出完美无缺的程序。建议先通过注释文字描述出问题求解思路的任务框架。然后，将其中比较容易的任务翻译成 C++ 代码。对于较难的任务，则可以假设存在一个或多个函数来完成这些任务，然后声明函数，并在后续工作中渐进地给出函数的定义，如此抽丝剥茧地逼近解决问题的终极目标。

此外，在编译和链接处理时，编译器会检查程序中的语法和符号链接错误，并给出错误提示信息。因此，代码的编辑和编译链接工作可以穿插进行，从而在编译器的帮助下快速发现并修正程序中的错误，加快源代码的编写工作。

4. 完整的程序代码

```
//functions.cpp
#include "stdafx.h"
#include <iostream>
using namespace std;
bool isprime(int);
int main() {
    int iNum;
    cin>>iNum;
    bool bFlag = isprime(iNum);
    if(bFlag) cout<<iNum<<"是素数"<<endl;
    else cout<<iNum<<"不是素数"<<endl;
    return 0;
}
```

```
//源文件: functions.cpp
#include "stdafx.h"
bool isprime (int num) {
    for(int i=2; i<num; i++) {
        if (0 == num % i) return false;
    }
}
```



```
    }  
    return true;  
}
```

图 11 大功告成的素数判定程序

疯狂实践系列二：自身和反序数均是素数

题目：求出 10~1000 之间满足如下条件的整数：它和它的反序数都是素数。例如，13 的反序数是 31，它们都是素数。

1. 训练目标

- (1) 理解自顶向下、分而治之的问题求解思维。
- (2) 学习并掌握 `for` 循环语句的用法。
- (3) 变量的定义 = 变量的数据类型 + 变量的名字。
- (4) 学习并掌握 `if` 语句的用法。
- (5) 学习并掌握 `while` 循环语句的用法。
- (6) 理解函数定义与声明的区别，能够把函数声明集中放到头文件中。

2. 编写问题求解的代码框架

暂缓编程，首先利用注释文字描述问题求解思路，形成求解问题的代码框架。

```
int main(){  
    //任务 1. 建立变量 i，让 i 从 10 变到 1000。  
    //任务 2. 判断每个 i 及其反序数是否素数，若是则输出 i。  
    return 0;  
}
```

然后，逐步把注释文字对应的任务 1 和任务 2 分别翻译成 C++ 程序代码。

3. 翻译任务 1：让 i 从 10 变到 1000。

对于任务 1，C++ 语言也提供了 `for` 循环语句，其语法形式为：

```
for (《初始化语句》; 《条件表达式》; 《修改表达式》 ) {  
    循环体  
}
```

其中，**初始化语句**，可以是声明语句或表达式，通常用于循环变量的初始化，只会被执行一次；**条件表达式**，在循环体每次执行之前，会对条件表达式进行求值测试，若求值为 **true** 则继续循环，若为 **false** 则停止循环；**修改表达式**，在循环体每次执行之后进行求值，通常用于修改循环变量，使得条件表达式朝着 **false** 的方向前进。

利用 **for** 语句，可以将任务 1 翻译为如下代码：

```
int main(){
    //任务 1. 建立变量 i，让从 10 变到 1000。
    for(int i = 10; i<=1000; i++) {
        //任务 2. 判断每个 i 及其反序数是否素数，若是则输出 i。
    }
    return 0;
}
```

4. 翻译任务 2：判定 i 及其反序数是否素数

任务 2 相对比较难翻译，但是我们可以假设存在一个函数可以判断 **i** 及其反序数是否素数，并且假设函数的名字叫做 **bothIsPrime**。当我们把 **i** 放到 **bothIsPrime** 函数中的时候，**bothIsPrime** 函数就会返回 **true** 或者 **false**，其中 **true** 表示 **i** 和 **i** 的反序数都是素数，否则就返回 **false**。即函数 **bothIsPrime** 的原型样貌如下：

```
bool bothIsPrime(int);
```

利用该函数可以很容易地将任务 2 翻译为如下代码：

```
bool bothIsPrime(int); //函数声明
int main(){
    //1. 建立变量 i，让从 10 变到 1000。
    for(int i = 10; i<=1000; i++) {
        //2. 对于每个 i，判定 i 及其反序数是否素数，若是则输出 i。
        if(bothIsPrime(i)) cout<<i<<endl;
    }
    return 0;
}
```

至此完成 `main` 函数。但是，在 `main` 函数中假设存在的 `bothIsPrime` 函数，实际上它并不存在。下一步就需要定义 `bothIsPrime` 函数。

5. 定义 `bothIsPrime` 函数

函数 `bothIsPrime`，用于判定给定的正整数及其反序数是否都是素数。同样，暂缓编程，利用注释描述 `bothIsPrime` 函数的代码框架。

```
/*
    函数名: bothIsPrime
    功能: 判定 m 及其反序数是否素数
    形式参数: int, 待判定的正整数 m
    返回值类型: bool, true 表示 m 及其反序数是素数, 否则 false
*/
bool bothIsPrime(int m){
    //1. 计算 m 的反序数 rm
    //2. 判定 m 和 rm 是否都是素数, 若是返回 true, 否则返回 false
}
```

任务 1 和任务 2 都不是太容易。我们继续假设：（1）假设存在函数 `invert` 可以求出一个整数的反序数；（2）假设存在函数 `isprime` 可以判定一个数是否素数。

如果上述两个假设成立的话，也就是 `invert` 和 `isprime` 函数存在，则任务 1 和任务 2 的翻译易如反掌。代码如下：

```
int invert(int);    //声明存在 invert 函数
bool isprime(int);  //声明存在 isprime 函数
bool bothIsPrime (int m){
    //1. 计算 m 的反序数 rm
    int rm = invert(m); //调用函数求出 m 的反序数
    //2. 判定 m 和 rm 是否都是素数, 若是返回 true, 否则返回 false
    if (isprime(m) && isprime(rm)) return true;
    return false;
}
```

至此完成 `bothIsPrime` 函数定义完成。但是，`bothIsPrime` 函数成立的前提是存在 `invert` 和 `isprime` 函数，不过二者并不存在，需要定义它们。

5.1 定义 `invert` 函数

函数 `invert`，用于求一个整数的反序数。假设求整数 `m=1247` 的反序数。求解思路是：假设反序数 `rm` 初始为 0，从 `m` 中取得其个位数并累加到 10 倍的反序数 `rm` 中，然后让 `m` 除以 10 消除掉个位数；反复执行该过程即可得到最终的反序数，如表 4 所示。

表 4 求 1247 的反序数的过程示意

序号	m (初始=1247)	m%10	反序数 rm (初始=0)
1	m=1247	7	rm=0*10+7=7
2	m=1247/10 = 124	4	7*10+4=74
3	m=124/10 = 12	2	74*10+2=742
4	m=12/10=1	1	742*10+1=7421
5	m=1/10=0		

将上述思路，用注释表达出来并翻译就形成如下的 `invert` 函数代码：

```
/*
    函数名: invert
    功能: 求 m 的反序数
    形式参数: int, 需要求反序数的正整数 m
    返回值类型: int, m 的反序数
*/
int invert(int m){
    //1. 定义变量 rm，最终求得的反序数存放在 rm 中。
    int rm = 0;
    //2. 反复从 m 中取得个位数，并将 m 设置为 m/10
    while(m > 0) { rm = rm*10 + m%10;    m = m / 10; }
    //3. 返回反序数 rm
    return rm;
}
```

5.2 定义 isprime 函数

函数 `isprime`，用于判定一个整数是否素数。在疯狂修炼系列一中已经实现，此处不再重复说明，直接给出代码：

```
bool isprime (int num) {
    for(int i=2; i<num; i++) {if (0 == num % i) return false;}
    return true;
}
```

6. 完整的程序代码

至此源程序编写完成（参见图 12），选择菜单【生成/生成解决方案】，或者使用快捷键 **F6**，进行源代码的编译链接工作，若无误即可执行程序，测试执行结果。

通过该程序的编写实践，应该能够体验到自顶向下、分而治之的问题求解思维。在编程实践中，不要奢望一次就编写出完美无缺的程序。建议先通过注释文字描述出问题求解思路的任务框架。然后，将其中比较容易的任务翻译成 **C++** 代码。对于较难的任务，则可以假设存在一个或多个函数来完成这些任务，然后声明函数，并在后续工作中渐进地给出函数的定义，如此抽丝剥茧地逼近解决问题的终极目标。

```
//test.cpp
```

```
#include "stdafx.h"
#include "function.h"
#include <iostream>
using namespace std;
int main() {
    for(int i = 10; i<=1000; i++) {
        if(bothIsPrime(i)) cout<<i<<endl;
    }
    return 0;
}
```

```
//function.h, 头文件实现函数集中声明
```

```
bool bothIsPrime(int n);
int invert(int m);
bool isprime (int num);
```

```
//functions.cpp
#include "stdafx.h"
#include "function.h"
bool bothIsPrime (int m) {
    int rm = invert(m); //调用函数求出 m 的反序数
    if (isprime(m) && isprime(rm)) return true;
    return false;
}
int invert(int m) {
    int rm = 0;
    while(m > 0) {rm = rm*10 + m%10;m = m / 10;}
    return rm;
}
bool isprime (int num) {
    for(int i=2; i<num; i++) {if (0 == num % i) return false; }
    return true;
}
```

图 12 完整的源程序代码

疯狂实践系列三：最大公约数和最小公倍数

题目：任意输入两个自然数，求它们的最大公约数和最小公倍数并输出。

1. 训练目标

- (1) 训练自顶向下、分而治之的问题求解思维。
- (2) 针对暴力搜索的问题求解思路进行编程。
- (3) 针对更为精巧的问题求解思路进行编程。
- (4) 熟练掌握如何准备数据：数据类型+变量名字。
- (5) 熟练掌握 C++ 语言提供的各类运算符和表达式。
- (6) 熟练掌握如何进行程序流程控制。

选择分支流程：if、if-else、switch、:?

循环流程：for、while、do-while

跳转流程：break、continue、return

- (7) 熟练掌握 cin 和 cout 进行键盘输入和屏幕输出。

2. 编写问题求解的代码框架

首先利用注释文字描述问题求解思路，形成求解问题的代码框架。针对该题目，输入两个自然数，求它们的最大公约数和最小公倍数并输出。main 函数代码如下：

```
int main(){
    //1.描述数据
    int m, n;
    //2.输入数据
    cin >> m >> n;
    //3.处理数据：假设存在 gcd 的函数可以求两数的最大公约数
    int gcd(int, int); //声明该函数存在
    int s = gcd(m, n); //调用函数 gcd 求最大公约数
    int t = m*n / s;    //最小公倍数等于 m*n/s
    //4.输出处理结果
```

```

    cout <<"gcd("<< m <<","<< n <<") = "<< s <<endl;
    cout <<"lcm("<< m <<","<< n <<") = "<< t <<endl;
    return 0;
}

```

gcd 是最大公约数的英文缩写 (greatest common divisor), lcm 是最小公倍数的英文缩写 (least common multiple)。main 函数完成之后, 剩下的问题就是制造求最大公约数的机器, 也就是定义 gcd 函数。

3. 定义 gcd 函数

gcd 函数用于求解两个整数的最大公约数, 其函数原型为:

```
int gcd(int n1, int n2);
```

最直接办法是暴力测试, 当然还有更精巧的方法。我们当下是在学习计算机语言, 无论求解思路笨拙或者精巧, 我们都要有能力给予实现。

3.1 暴力测试法实现 gcd

暴力测试法的问题求解思路是: 对于给定两个整数 n1 和 n2, 假设 n1 较小, 则我们可以从 n1 到 1 一个个地试验, 倘若遇到一个能够整除 n1 且能够整除 n2 的, 那么这个数就是最大公约数。翻译成 C++ 语言就是如下的函数定义:

```

int gcd(int n1, int n2) {
    //让变量 i 从两个数中较小的数开始一直变到 1
    for (int i = n1 < n2 ? n1 : n2; i >= 1; i--) {
        //如果 i 能够整除 n1 和 n2, 则 i 就是最大公约数, 返回 i
        if ( 0 == n1 % i && 0 == n2 % i) return i;
    }
}

```

3.2 欧几里德算法实现 gcd

暴力测试法可以解决问题, 但是效率不高。如果 n1 和 n2 非常大, 则程序执行需要耗费较长的时间。实际上, 关于 n1 和 n2 的最大公约数有一个重要的定理。

定理： n_1 和 n_2 的最大公约数，等于 n_2 和 $n_1 \% n_2$ 的最大公约数。即：

$$\text{gcd}(n_1, n_2) = \text{gcd}(n_2, n_1 \% n_2) \quad (\text{证明略})$$

这种最大公约数求解思路最早出现在欧几里德《几何原本》中，称为欧几里德算法或者辗转相除法。如下函数就通过辗转相除法求最大公约数：

```
//辗转相除法求最大公约数
int gcd(int n1, int n2){
    while(n2 > 0) { //不断进行辗转相除运算，直到 n2 变成 0
        int temp = n1 % n2;    n1 = n2;    n2 = temp;
    }
    return n1; //最后，n1 就是最大公约数，返回 n1
}
```

函数的基本思路是，如果 n_2 大于 0，就循环反复地将 n_1 变成 n_2 ， n_2 变成 $n_1 \% n_2$ 。循环停止时 n_2 必定会变成 0，此时的 n_1 就是最大公约数。

3.3 欧几里德算法递归实现 gcd

求最大公约数的函数，就可以通过递归调用的方式来实现。如果存在一个函数 `gcd`，可以求得 n_1 和 n_2 的最大公约数，即 `gcd(n_1 , n_2)`；那么，使用这个函数 `gcd`，必定可以求得 n_2 和 $n_1 \% n_2$ 的最大公约数，即 `gcd(n_2 , $n_1 \% n_2$)`，并且这两个最大公约数相等。根据这一思路，可以描述出如下的程序代码：

```
int gcd(int n1, int n2) {
    //如果 n2 为 0，则最大公约数就是 n1；
    if (n2 == 0) return n1;
    //否则，最大公约数就等于调用 gcd(n2, n1 % n2) 的执行结果。
    return gcd(n2, n1 % n2);
}
```


疯狂实践系列四：进制转换 10to2

题目：编写程序演示将一个 10 进制整数转换为 2 进制的转换过程。已知，将一个 10 进制整数转换为 2 进制数的运算规则是：除 2 取余。例如，

$$(142) = (10001110)_2。$$

要求程序在屏幕上输出如右图所示的转换过程。

```
2 | 142
  -----
2 | 71...0
  -----
2 | 35...1
  -----
2 | 17...1
  -----
2 | 8...1
  -----
2 | 4...0
  -----
2 | 2...0
  -----
2 | 1...0
  -----
  0...1
```

1. 训练目标

- (1) 训练自顶向下、分而治之的问题求解思维。
- (2) 熟练掌握如何准备数据：数据类型+变量名字。
- (3) 熟练掌握 C++ 语言提供的各类运算符和表达式。
- (4) 熟练掌握如何控制程序流程：

选择分支流程：if、if-else、switch、:?

循环流程：for、while、do-while

跳转流程：break、continue、return

- (5) 熟练掌握 cin 和 cout 进行键盘输入和屏幕输出。
- (6) 掌握函数的就地声明、集中声明和头文件声明。

2. 自顶向下分治

根据结构化编程的分治思想，自顶向下开始编程描述解题思路，首先应该编写 main 函数的代码。main 函数不外乎进行数据输入、处理和输出任务。

```
int main( ) {
    int n;
    cout << "输入 10 进制整数: ";
    cin >> n;
    //假设存在函数 dec2bin 可以输出转换过程
    void dec2bin(int); //声明函数存在
    dec2bin(n); //调用函数完成转换过程输出
    return 0;
```

```
}
```

显然，子问题“输出转换过程”的编程比较复杂，一时难以完成。因此，我们假设存在如下的 `dec2bin` 函数，可以完成“输出转换过程”的工作，

```
void dec2bin(int); //声明函数存在
```

因为该函数目前还不存在，所以，我们通过函数声明向编译器保证函数未来存在。无论当下还是未来，只要该函数存在，那么，求解整个问题就易如反掌，或者说编写 `main` 函数易如反掌，不过是调用 `dec2bin` 函数而已，如图 13 所示。

至此，在 `dec2bin` 函数假设存在的前提下，实现了最顶层整个问题的求解。但是，`dec2bin` 函数目前并不存在。因此，问题就细化分解为一个子问题：“输出转换过程”，也就是说，接下来需要给出函数 `dec2bin` 的定义。

2.1 第一层子问题：dec2bin 函数

函数 `dec2bin` 的任务是输出转换过程。分析转换过程演示可以发现，转换过程的输出总体上需要三个步骤：

- (1) 首先在屏幕上输出转换过程的首项；
- (2) 然后循环输出中间的多个转换过程项；
- (3) 最后输出一个转换过程的尾项。

因此，在 `dec2bin` 函数中，通过函数声明假设存在三个函数：`printhead`、`printitem`、`printfoot`，分别用于输出转换过程的首项、中间过程项、以及尾项，然后调用这三个函数完成 `dec2bin` 函数的编写工作（如图 13 所示），代码如下：

```
void dec2bin( int n ) {
    //假设存在 printhead 函数可以输出过程首项
    void printhead(int); //声明函数存在
    printhead(n); //调用函数输出过程首项
    while(n > 1) {
        //假设存在 printitem 函数可以输出中间过程项
        void printitem(int, int); //声明函数存在
        printitem(n/2, n%2); //调用函数输出中间过程项
        n = n / 2;
    }
}
```

```

    }
    //假设存在 printfoot 函数可以输出过程尾项
    void printfoot(); //声明函数存在
    printfoot(); //调用函数输出过程尾项
}

```

至此，我们就是将第一层的子问题“输出转换过程”，向下分解细化成第二层的三个子问题：输出过程首项、输出过程中间项、输出过程尾项。

2.2 第二层子问题：printhead、printitem、printfoot 函数

接下来是提供 printhead、printitem、printfoot 三个函数的定义。

2.2.1 输出转换过程的首项：printhead 函数

```

//输出过程首项，参数 n 是输入的 10 进制数
void printhead( int n ) {
    //假设存在函数 printtab，输出缩进
    void printtab(); //声明函数存在
    printtab(); //调用函数输出缩进
    cout << "2|\t" << n << endl;
    printtab(); //调用函数输出缩进
    //假设存在函数 printsep，输出分割线
    void printsep(); //声明函数存在
    printsep(); //调用函数输出分割线
}

```

在 printhead 函数中，假设存在函数 printtab 和 printsep，用于输出缩进和分割线，我们调用了这两个函数来实现 printhead。

2.2.2 输出转换过程中间项：printitem 函数

```

//输出转换过程中间项，参数 n 为商项，参数 m 为余数项
void printitem ( int n , int m ) {
    void printtab(); //声明函数存在

```

```

    printtab(); //调用函数输出缩进
    cout << "2|\t" << n << "\t....." << m << endl;
    printtab(); //调用函数输出缩进
    void printsep(); //声明函数存在
    printsep(); //调用函数输出分割线
}

```

在 `printitem` 函数中，同样假设存在函数 `printtab` 和 `printsep`，用于输出缩进和分割线，我们调用这两个函数实现 `printitem`。

2.2.3 输出转换过程的尾项：printfoot 函数

```

//输出转换过程的尾项: 0 ... 1
void printfoot () { //输出转换过程的尾项
    void printtab(); //声明函数存在
    printtab();
    cout << "\t0" << "\t.....1" << endl;
}

```

在 `printfoot` 函数中，需要重复调用函数 `printtab`，向屏幕输出缩进。可见，这三个函数最后都归结为 `printtab`、`printsep` 函数的调用，如图 13 所示。也就是说，这三个子问题的求解被细化分解为第三层的两个子问题：输出缩进和分割线。

2.3 第三层子问题：printtab、printsep 函数

第三层的两个子问题是输出缩进和输出分割线，其编码易如反掌。

```

inline void printtab() { // 输出缩进
    cout << "\t\t\t";
}
inline void printsep() { // 输出分割线
    cout << " -----" << endl;
}

```


这两个函数代码简单且被多次调用，故不妨把它们定义为内联函数（`inline`）²。至此，整个问题完全得解，编译连接执行，屏幕上就会输出转换过程演示。

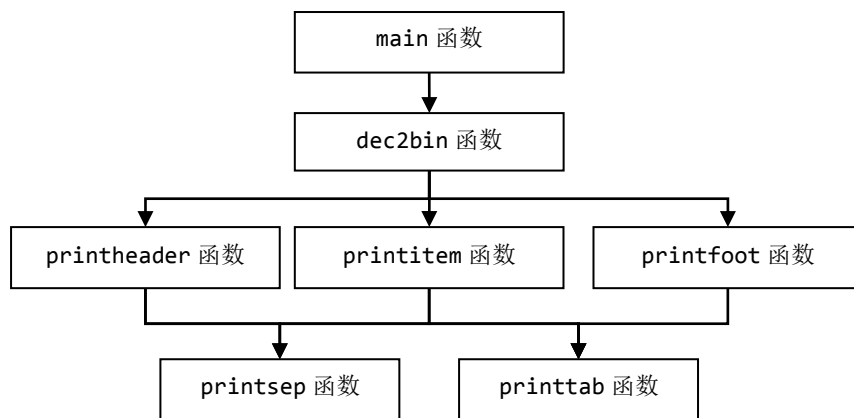


图 13 进制转换演示程序的函数调用关系

在上述程序的编写过程中，每当我们假设存在一个函数时，立刻就地进行函数声明将该函数引入进来，这是一种写到哪里就声明到哪里的做法，可以称做“就地声明”。就地声明，非常适合于编写规模较小的“玩具”程序。

3. 函数集中声明

“就地声明”存在诸多问题，首先是声明被限制在函数体的局部作用域，第二是声明散乱分布在函数代码中不易管理。需要把声明放在所有函数的外面，使得声明在整个文件可见。对于规模较小的程序，通常在源文件开始位置集中进行函数声明。

```
//dec2bindemo.cpp
#include <iostream>
using namespace std;
//函数声明部分
void dec2bin(int);
void printhead(int);
void printitem(int, int);
void printfoot();
void printtab();
void printsep();
```

²注意把内联函数 `printtab` 和 `printsep` 的定义放在其调用之前，如放在 `main` 函数之前。

```
//函数定义部分
int main( ) { ... }
void dec2bin( int n ) {...}
void printhead ( int n ) { ...}
void printitem ( int n , int m) { ... }
void printfoot () { ... }
inline void printtab() { ... }
inline void printsep() { ... }
```

这里，函数声明没有放在任何函数体内部，而是放在源文件的开始位置，之后这些函数名就在整个文件中可见，在其他所有函数中均可调用。

4. 头文件集中声明

对于规模较大的程序，需要多人组团协作开发。此时，每个团队成员仅仅负责部分程序代码的编写工作，倘若每个人都在各自源文件的开始位置进行集中声明，同样会导致函数声明分散难以管理，难以保证声明的一致性。

C++语言为此引入了另外一种集中声明方式——**头文件声明方式**。

头文件声明，就是将所有函数声明统一放到一个或几个文件中，这种文件称为头文件（**header file, *.h**），区别于源文件（**source file, *.cpp**）。然后，在需要声明的源文件中通过**#include** 指令将头文件包含进来。

4.1 定义头文件

定义头文件，就是生成一个独立的文件，文件扩展名通常为***.h**，在其中放置函数或者变量的声明。对于进制转换程序，可以定义如下头文件：

```
//头文件：myfunc.h，把函数声明集中放置
//函数声明部分
void dec2bin(int); //输出十进制整数转换为二进制的转换过程
void printhead(int); //输出转换过程首项
void printitem(int, int); //输出转换过程项
void printfoot(); //输出转换过程尾项
inline void printtab() { ... } //输出缩进，内联函数需包含定义
inline void printsep() { ... } //输出分割线，内联函数需包含定义
```

在头文件 `myfunc.h` 中，将所有的函数声明统一集中收纳起来。当然，头文件中放置的只是函数声明，而函数声明只是宣告函数的存在性，给出函数样貌的原型说明³，而函数的具体实现需要在函数定义中给出。也就是说，在头文件 `myfunc.h` 中声明的函数，需要在源文件（`*.cpp`）中给出定义⁴，例如：

```
//源文件: myfunc.cpp
#include "myfunc.h"
#include <iostream>
#include "myfunc.h"          //包含头文件进行集中声明
using namespace std;
void dec2bin( int n ) { ... }
void printhead ( int n ) { ... }
void printitem ( int n , int m) { ... }
void printfoot () { ... }
```

函数声明放置在头文件中，相应的函数定义则放置在源文件中。显然，函数的声明和定义应该是成对匹配的，进而头文件和源文件也应该是成对匹配的。

通常约定，头文件与相应的源文件，应该具有相同的主文件名，二者的区别在于文件扩展名不同。例如：头文件 `myfunc.h` 和源文件 `myfunc.cpp`。

4.2 包含头文件

当头文件定义完成之后，如果在编写程序时需要声明头文件中的函数，则只需要把该头文件通过 `#include` 指令包含进源文件中即可。

`#include` 是一种预编译指令，用于在预编译处理时包含头文件。使用方法如下：

```
#include <头文件名>    //包含标准头文件
#include "头文件名"    //包含用户头文件
```

对于 `#include` 指令，C++编译器在预编译处理时会查找 `#include` 指令之后的头文件，找到后就用该头文件中的内容替换掉 `#include` 指令。实际上就是将头文件中的声明，拷贝粘贴到 `#include` 指令的位置，这相当于完成了函数声明。

³注意，内联函数的定义需放在头文件中，以便在编译期间找到函数定义实现调用代码展开。

⁴内联函数 `printtab` 和 `printsep` 的定义放在头文件中，源文件中不能重复定义。

（1）尖括号包含方式

如果`#include` 指令后面是尖括号括起来的头文件，则编译器就会认为该头文件是标准头文件，是由语言系统本身或者外部程序库提供的，此时，编译器就会从编译系统预先定义好的文件目录开始查找这个头文件；

（2）双引号包含方式

如果`#include` 指令后面是双引号括起来的头文件，则编译器认为该头文件是用户自己定义的头文件，通常放在程序的当前目录中，从当前目录开始查找。

例如，在 `myfunc.cpp` 中，`dec2bin` 函数调用了 `printhead`、`printitem` 和 `printfoot`，应该对这些函数进行声明。由于头文件 `myfunc.h` 中包含了所需的函数声明，现在只要把该头文件包含进源文件 `myfunc.cpp` 中即可完成声明：

```
#include "myfunc.h"
```

因为 `myfunc.h` 是我们自定义的头文件，故选择双引号的头文件包含方式。头文件包含之后，在 `dec2bin` 函数的定义中，我们就可以直接调用 `printhead`、`printitem` 和 `printfoot` 函数，而毋须反复地进行函数声明。

进制转换演示程序中 `main` 函数所在源文件 `dec2bindemo.cpp` 中，需要调用函数 `dec2bin`，同样通过`#include` 指令完成函数声明。示例如下：

```
//dec2bindemo.cpp
#include <iostream>
using namespace std;
#include "myfunc.h"
//函数定义部分
int main( ) { ... }
```

总之，通过头文件和包含指令可以将所有的声明集中起来统一管理和使用，保证共享的声明只有一个公共版本，修改声明时只需要修改头文件即可，适合于规模较大程序的协作编写，每个程序员都必须掌握这种高效方便的声明方法。

4.3 预编译头文件

一个规模很大的程序，编译一次需要消耗很久的时间。为了节省时间，把程序中一些稳定不变的代码单独放置到一个头文件中做一次编译，之后就不再动它。包含该头文件的

程序代码，在编译时不会重复编译该头文件的代码，从而节省了大量的编译时间，这种头文件叫做预编译头文件，典型预编译头文件如 `stdafx.h`。

疯狂实践系列五：谁参加了会议

题目：假设有 ABCDE 五名同学，他们都有可能参加了某次会议，根据下列条件判断哪些人参加了会议：（1）若 A 参加则 B 也参加；（2）B 和 C 不能同时参加；（3）C 和 D 或者都参加，或者都不参加；（4）D 和 E 至少有一人参加；（5）如果 E 参加，则 A 和 D 也都参加。尝试编写程序找到答案。

1. 训练目标

- （1）理解自顶向下、分而治之的问题求解思维。
- （2）构造问题的解空间，在解空间搜索的问题求解策略。
- （3）理解掌握位串运算，数字与字母 ASCII 编码的转换。

2. 问题求解思路：构造解空间

类比一下之前的问题：找出 1000 以内所有素数。当时，我们并不知道哪个数是素数，因此假设所有数都有可能是素数，然后针对每个可能的数进行是否素数的判定。在计算机科学领域中，这是一种非常普遍的思维方式，先找到所有可行解的集合（称为解空间），然后针对每个可行解进行条件判定。

推广这种思维方式，首先寻找五个同学是否参加会议的所有可行解。很直观，第 1 个可行解是所有人都不参加，第 2 个可行解是 E 参加其他人不参加，第 3 个可行解是 D 参加其他人不参加，……，最后一个可行解是所有人都参加。

只需要一点点排列组合的知识，就可以得出共有 $2^5=32$ 个可行解。如果用二进制 0 表示不参加，1 表示参加，则所有可能解表示成如下的解空间：

A	B	C	D	E	十进制值
0	0	0	0	0	= 0
0	0	0	0	1	= 1
0	0	0	1	0	= 2
...	= ...
1	1	1	1	0	= 30
1	1	1	1	1	= 31

对应的 10 进制的解空间是：0~31。0 表示所有人都不参加，31 表示所有人都参加。

问题求解就针对解空间中每个可行解，测试其是否满足题目给出的条件。

3. 自顶向下分治

3.1 第一层问题：main 函数

根据上文的分析过程尝试编写 `main` 函数，很简单，搞不掂的问题就假设存在一个函数可以帮你搞掂。示例代码如下：

```
const int N = 5;    //假设有 N 个同学，从字母'A'开始编号
int main() {
    for(int i = 0; i<(1<<N); i++) {
        //判断可行解 i 是否是真实解
        if(IsAnswer(i)) //假设函数 IsAnswer 判断是否真实解
            print(i);    //假设函数 print 可以输出结果
    }
    return 0;
}
```

这里 `1<<N` 表示将 1 左移 N 位，结果就是 2^N 。`main` 函数中假设存在的 `IsAnswer` 函数和 `print` 函数，实际并不存在，所以记得要做函数声明的工作。

3.2 第二层问题：定义 `IsAnswer` 和 `print` 函数

现在问题分解归结为 `IsAnswer` 函数和 `print` 函数的定义。其中，

(1) `IsAnswer`：用于判断可行解 `a` 是否满足题目条件。其原型为：

```
bool IsAnswer(int a);
```

(2) 函数 `print`：用于输出一个给定的解 `a`。其函数原型为：

```
void print(int a);
```

3.2.1 定义 `IsAnswer` 函数

给定一个可行解 `a`，我们需判断该可行解是否满足题目给出的 5 个命题条件。假设存在 5 个函数：`hit1`~`hit5`，分别用于判断可行解 `a` 是否满足 5 个命题。

```
bool IsAnswer(int a) {
    if(hit1(a) && hit2(a) && hit3(a) && hit4(a) && hit5(a))
```



```
    return true;
    return false;
}
```

问题经过分治求精处理，越来越接近本质。接下来定义 5 个命题判定函数。

(1) hit1: 若 A 参加则 B 也参加

分析之下，可以推断：只有当 A 参加 B 没有参加时该命题取值 **false**，其他情况取值为 **true**。注意，当 A 没有参加时 B 参加或不参加，命题都成立。代码如下：

```
bool hit1(int a) {
    if (join(a, 'A') && !join(a, 'B')) return false;
    return true;
}
```

这里，假设存在函数 `join`，能够判断 `a` 表示的解中某人是否参加了会议。例如，判断解 `a` 中 'B' 是否参加了，就调用 `join(a, 'B')`。至于 `join` 函数的定义，先不要急，放松一下，这里只要保证有函数声明就可以了。

(2) hit2: B 和 C 不能同时参加

逻辑很简单，B 和 C 同时参加命题取值 **false**，否则为 **true**。代码如下：

```
bool hit2(int a) {
    if (join(a, 'B') && join(a, 'C')) return false;
    return true;
}
```

(3) hit3: C 和 D 或者都参加，或者都不参加

```
bool hit3(int a) {
    if (join(a, 'C') && join(a, 'D')) return true;
    if (!join(a, 'C') && !join(a, 'D')) return true;
    return false;
}
```

(4) hit4: D 和 E 至少有一人参加

```
bool hit4(int a) {
    return join(a, 'D') || join(a, 'E');
}
```

(5) hit5: 如果 E 参加，则 A 和 D 也都参加

与命题 1 类似，只有当 E 参加而 A 和 D 没有都参加时该命题取值 **false**，其他情况取值为 **true**。

```
bool hit5(int a) {
    if (join(a, 'E') && (!join(a, 'A') || !join(a, 'D')))
        return false;
    return true;
}
```

3.2.2 定义 print 函数

给定一个可行解 **a**，我们需根据 **a** 的二进制位串取值输出相应同学的代号以及该同学的参加情况，这正是假设的 **print** 函数的职责。尝试编写代码如下：

```
void print(int a) {
    for(int i=0; i<N; i++) {
        char id = 'A' + i;    //产生每个人的代号
        if(join(a, id)) cout<<id<<"参加了"<<endl;
        else cout<<id<<"没参加"<<endl;
    }
}
```

至此，所有的问题最后都归结到调用 **join** 函数，似乎看到了胜利的曙光。

3.3 最后的堡垒：定义 join 函数

我们期待的 **join** 函数，是对于解 **a** 表示的各位同学的参加状态，判断某个同学是否参加，参加返回 **true**，没参加返回 **false**。实际上就是测试 **a** 的二进制位串中相应位置是 0 还是 1，0 没参加，1 参加。

假设 **a = 22**，则二进制位串为 **10110**，如右图所示。A 对应的位置为 1，表示在解 **a** 表示的状态中 A 参加了，类推地，B 没参加，CD 参加了，E 没参加。

	A	B	C	D	E
a	1	0	1	1	0
&	1	0	0	0	0
=	1	0	0	0	0

现在想知道'A'同学有没有参加，可以将整数 1 左移 4 位，得到二进制位串：10000，然后用该位串与 a 做按位与运算，如果运算结果大于 0，则'A'位置对应的二进制数是 1，即'A'参加了，否则'A'位置对应的二进制数是 0，'A'没参加。翻译代码如下：

```
bool join(int a, char id) {  
    return a & (1 << (N - (id - 'A') - 1));  
}
```

这里，'A'左移的位数是 4，'B'左移的位数是 3，……，'E'左移的位数是 0。可以通过公式 $(N - (id - 'A') - 1)$ 算出字符 id 需要左移的位数。

3.4 完整代码组织结构

将上文的思维分析迭代过程梳理一下，使用头文件技术分割源代码，实现多文件程序的组织管理。回过头来想想，完美程序是给计算机用的，是给别人看的，形式完美的代码背后是我们筚路蓝缕的探索发现过程，华美的睡袍里面爬满了虱子。

//头文件: join.h

```
const int N = 5;    //假设是 N 个同学  
bool join(int, char);  
void print(int);  
bool IsAnswer(int);  
bool hit1(int);  
bool hit2(int);  
bool hit3(int);  
bool hit4(int);  
bool hit5(int);
```

//源文件: join.cpp

```
#include "stdafx.h"  
#include "common.h"  
#include <iostream>  
using namespace std;  
inline bool join(int a, char id) { ... }  
void print(int a) { ... }  
bool IsAnswer(int a) { ... }  
bool hit1(int a) { ... }  
bool hit2(int a) { ... }  
bool hit3(int a) { ... }  
bool hit4(int a) { ... }
```

```
bool hit5(int a) { ... }
```

```
//源文件: joinproblem.cpp
```

```
#include "stdafx.h"
```

```
#include "join.h"
```

```
int main(){
```

```
    for(int i = 0; i<(1<<N); i++) { if(IsAnswer(i)) print(i);} 
```

```
    return 0;
```

```
}
```

疯狂实践系列六：约瑟夫问题之还有谁

题目：n 个人围成一圈，顺序排号。从第一个人开始报数（从 1 到 3 报数），凡报到 3 的人退出圈子，问最后留下的是原来的第几号人物。

1. 训练目标

- (1) 理解自顶向下、分而治之的问题求解思维。
- (2) 掌握数组、动态数组及数组用作函数参数的语法。
- (3) 了解常用程序调试方法，掌握 VS2010 的调试工具。
- (4) 分析问题的数学递推规律，可以编写出更高效的程序。
- (5) 无论解题思路多么笨拙，我们都能自如地将其编程表达。

2. 自顶向下分治

可以采用非常直观的模拟游戏法来求解约瑟夫问题。把 1~n 的编号放到一个数组中，然后反复遍历数组模拟报数过程，每报数到 3 就把该位置的编号置为 0；模拟报数过程 n-1 次之后，圈里面必定只剩一个人，此时输出这个人的编号。

2.1 第一层问题：main 函数

根据模拟报数的问题求解思路，尝试编写 main 函数如下：

```
int main() {  
    int n = 0, m = 0;  
    cout<<"请输入游戏的人数 n (n>=2): ";cin>>n;  
    cout<<"请输入报数步距 m (m>=1): ";cin>>m;  
    int* ring = buildRing(n); //建立圈子  
    play(ring, n, m); //模拟报数游戏  
    cout<<"报数游戏最后剩下的人是: ";  
    print(ring, n); //输出游戏结果  
    deleteRing(ring); //释放圈子
```

```
    return 0;
}
```

程序的逻辑过程非常简单，首先输入人数和报数步距，然后建立 1~n 的圈子，模拟报数游戏，输出游戏结果，释放圈子。问题归结为如下四个函数的定义：

```
int* buildRing(int);           //建立 1~n 的圈子
void deleteRing(int*);        //模拟报数游戏
int play(int*, int, int);      //输出游戏结果
void print(int*, int);         //释放圈子
```

柿子捡软的捏，四个函数中 buildRing、deleteRing、print 比较简单，先搞掂。

2.1.1 第二层问题：定义 buildRing 函数

buildRing 函数负责建立圈子：建立大小为 n 的动态数组，然后将编号 1~n 填充进去，最后返回动态数组的指针。代码如下：

```
int* buildRing(int n) {
    int *ring = new int[n];
    for(int i=0; i<n; i++) ring[i] = i+1;
    return ring;
}
```

2.1.2 第二层问题：定义 deleteRing 函数

deleteRing 函数负责释放圈子对于的动态数组。代码如下：

```
void deleteRing(int *ring) { delete[] ring; }
```

2.1.3 第二层问题：定义 print 函数

print 函数负责输出游戏结果，即从圈子中找到编号不为 0 的人输出。代码如下：

```
void print(int* ring, int n) {
    for(int i=0; i<n; i++) {
        if(ring[i]!=0) { cout<<ring[i]; break; }
    }
}
```

2.2 难啃的骨头：定义 play 函数

play 函数负责模拟报数游戏，比较难啃。让我们先理清基本思路，然后进行自顶向下的分治：整个报数游戏是由 $n-1$ 个轮次的报数组成的；每个轮次的报数过程非常固定，从上次被踢出人的下一位置开始，向前报数 m 个人，踢出第 m 个人。

可以假定存在 playRound 函数能够完成一个轮次的报数。函数原型为：

```
//start 表示报数的起点（上次报数被踢出人的下一位置）  
void playRound(int* ring, int n, int m, int& start);
```

每次报数完成后，需将 start 的值更新为被踢出人的下一位置，用做下一轮报数的起点。所以，playRound 函数中 start 参数声明为引用类型。有了 playRound 函数，play 函数的定义变得易如反掌。代码编写如下：

```
void play(int* ring, int n, int m) { //模拟 n-1 轮次报数  
    int start = 0;  
    for(int i=0; i<n-1; i++) playRound(ring, n, m, start);  
}
```

至此，问题归结为 playRound 函数的定义。

2.2.1 一轮报数：定义 playRound 函数

playRound 函数模拟一个轮次的报数，每次报数以 start 为起点报数 m 次，将第 m 个人踢出圈子。将 playRound 函数的功能表达为程序处理逻辑就是：

- (1) 报数：以 start 为起点循环扫描数组 ring，找到 m 个不为 0 的数组元素；
- (2) 踢人：将第 m 个数组元素置为 0。

根据上述程序处理逻辑，尝试编写出如下的框架性代码：

```
void playRound(int* ring, int n, int m, int& start) {  
    //1. 报数  
    while(m>0) {  
        //1.1 每报数一次，m 减 1  
        //1.2 每循环一次，start 加 1  
    }  
    //2. 踢人
```

```

    ring[?] = 0;  //?处填充第 m 个人的位置
}

```

继续迭代精化上面的框架性代码，所谓报数一次，指的是在 `ring` 数组中找到一个不为 0 的数组元素，故将任务 1.1 翻译可得如下代码：

```

void playRound(int* ring, int n, int m, int& start) {
    //1. 报数
    while(m>0) {
        //1.1 每报数一次，m 减 1
        if(ring[start]!=0) m--;
        //1.2 每循环一次，start 加 1
        start++;
    }
    //2. 踢人
    ring[?] = 0;  //?处填充第 m 个人的位置
}

```

在 `while` 循环中，`m` 用于控制报数次数，每次报数 `m` 减 1，当 `m` 等于 0 时退出循环，即报数 `m` 次。`start` 用于扫描 `ring` 数组，每访问一个数组元素之后就加 1。

随着循环的执行 `start` 不断加 1，当 `start` 等于 `n-1` 时到达数组末尾，再加 1 就会超出数组下标范围。根据题意，此时 `start` 应该返回数组开始位置，继续扫描。只有这样，才能用一维数组模拟出一个环形的圈子。故在 `start++` 之后增加代码：

```

void playRound(int* ring, int n, int m, int& start) {
    //1. 报数
    while(m>0) {
        //1.1 每报数一次，m 减 1，start 加 1
        if(ring[start]!=0) m--;
        start++;
        if(start == n) start -= n;  //到数组末尾则回到开头
    }
    //2. 踢人
    ring[?] = 0;  //?处填充第 m 个人的位置
}

```



```
}
```

m 次报数之后，`start` 指向第 m 个人后面的那个位置。因此可以推断：第 m 个人的位置是 `start-1`。在程序代码?处填入 `start-1`。程序功能表达完成，不妨进一步给程序做一下美容，如把 `start++` 和 `if` 语句合并等等。最终代码如下：

```
void playRound(int* ring, int n, int m, int& start) {  
    while(m>0) {  
        if(ring[start++]!=0) m--;  
        start = start % n;//这种处理方式更简洁  
    }  
    ring[start-1] = 0;  
}
```

2.2.2 程序执行有 BUG

至此，代码编写完成，编译链接执行。输入 3 人，报数步距 1，执行结果如下：

```
请输入游戏的人数 n (n>=2): 3  
请输入报数步距 m (m>=1): 1  
报数游戏最后剩下的人是: 3
```

屏幕输出符合预期，大功告成。

兴奋之余再试试其他的输入： $n=4$ ， $m=3$ 。回车，执行，结果咣地一声巨响，跳出一个“Debug Error!”窗口，屏幕上有如下输出：

```
请输入游戏的人数 n (n>=2): 4  
请输入报数步距 m (m>=1): 3  
报数游戏最后剩下的人是: 1
```

多次尝试其他输入的情况，时不时会跳出 Debug Error! 窗口。程序有 BUG。

根据 BUG 窗口信息，程序在执行过程中发生了堆内存破坏（Heap Corruption）。所谓堆内存破坏，是因程序逻辑错误导致堆内存被意外改写。这种错误危害很大且时隐时现非常隐蔽，导致程序错误很难发现排查。

3. 通过调试定位 BUG

定位程序错误的原理很简单：观察程序执行过程中变量或者对象状态的变化是否符合预期。我们都知道，执行程序就是对变量或者对象进行加工处理，使其状态发生一系列改变并最终达到预期状态的过程。因此，在程序执行过程中，我们便可以通过各种手段和工具观察变量或者对象的一系列中间状态，观察这些状态是否符合加工处理的预期，倘有某个中间状态不符合预期，则导致该中间状态的代码或许存在问题。

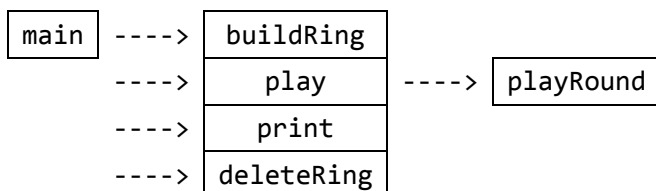
常用的观察变量或者对象中间状态的办法有：（1）打印输出；（2）使用断言 `assert`；（3）借助调试工具。

3.1 朴素调试技术：打印输出

打印输出，是把关注的变量或对象的状态输出在屏幕上，然后观察屏幕输出判断程序出错的位置。以报数游戏程序为例，演示打印输出这种朴素的程序调试技术。

3.1.1 根据函数调用关系粗略定位 BUG

报数游戏程序，目前总共有 6 个函数，函数之间的调用关系如下：



观察这 6 个函数的程序代码行数和程序处理逻辑，可以推断出如下结论：

（1）`main` 函数依赖于 `buildRing`、`play`、`print`、`deleteRing` 四个函数，只要这四个函数正确，基本上可以打包票 `main` 函数不会有问题。

（2）`buildRing`、`print`、`deleteRing` 三个函数代码很少，逻辑非常简单，出问题的可能性很小。

（3）`play` 函数代码只有两行，逻辑非常简单，除非 `play` 函数依赖的 `playRound` 函数有问题，否则根本不可能出错。

所以基本可以确定，BUG 疑凶应该就隐藏在 `playRound` 函数中。在自顶向下分治思维下编写程序，会将整个程序分解成环环相扣的一系列比较小的函数及其调用，较小的函数逻辑也就比较简单，出错的可能性就小，定位代码错误也更容易。

3.1.2 观察 playRound 函数执行过程

观察 `playRound` 函数的执行过程，实际上就是观察 `ring` 数组的变化情况。通过在屏幕上打印输出每次报数之前和之后的 `ring` 数组内容即可。

(1) 编写打印输出的辅助函数 `printRing`。

编写一个辅助函数 `printRing`，用于在屏幕上输出 `ring` 数组的状态：

```
void printRing(int* ring, int n) {
    for(int i=0; i<n; i++) cout<<ring[i]<<"\t";
    cout<<endl;
}
```

(2) 在 `playRound` 函数中调用 `printRing` 函数。

在 `playRound` 函数的开始和结尾处分别调用 `printRing` 函数。代码如下：

```
void playRound(int* ring, int n, int m, int& start) {
    cout<<"报数前状态: "; printRing(ring, n);
    while(m>0) {
        if(ring[start++]!=0) m--;
        start = start % n;
    }
    ring[start-1] = 0; //踢出
    cout<<"报数后状态: "; printRing(ring, n);
    cout<<endl;
}
```

(3) 观察 `printRing` 函数的打印输出结果。

编译链接执行程序，输入人数 `n=4`，步距 `m=3`，堆内存破坏错误依旧，但是在屏幕上输出了每次报数圈子数组的状态，如图 14 所示。

```
请输入游戏的人数 n (n>=2): 4
请输入报数步距 m (m>=1): 3
报数前状态: 1    2    3    4
报数后状态: 1    2    0    4
报数前状态: 1    2    0    4
```

报数后状态:	1	0	0	4
报数前状态:	1	0	0	4
报数后状态:	1	0	0	4
报数游戏最后剩下的人是: 1				

图 14 程序执行过程中 ring 数组的状态变化情况

观察图 14 中 ring 数组的变化情况, 可以看出: 第 1 轮和第 2 轮报数没有问题。

第 3 轮报数出了问题: 期待的编号为 4 的人被踢出, 即 `ring[3]=0` 并没有发生。继续推断, 在执行 `ring[start-1]=0` 时, `start-1` 不等于 3。

(4) 观察数组下标 start-1 的状态变化情况

既然 `start-1` 应该等于 3 却不等于 3, 那么它到底等于多少呢? 打印输出一下。

```
void playRound(int* ring, int n, int m, int& start) {
    cout<<"报数前状态: "; printRing(ring, n);
    while(m>0) {
        if(ring[start++]!=0) m--;
        start = start % n;
    }
    cout<<"start-1="<<start-1<<endl;
    ring[start-1] = 0; //踢出
    cout<<"报数后状态: "; printRing(ring, n);
    cout<<endl;
}
```

编译链接执行, 程序输出结果如图 15 所示。可以看出, 在第 3 轮报数时 `start-1` 的结果是 -1, 即第 `m` 个人的位置在 -1 这个位置, 执行 `ring[-1]=0`。

请输入游戏的人数 n (n>=2): 4				
请输入报数步距 m (m>=1): 3				
报数前状态:	1	2	3	4
start-1=2				
报数后状态:	1	2	0	4
报数前状态:	1	2	0	4
start-1=1				
报数后状态:	1	0	0	4

```
报数前状态: 1    0    0    4
start-1=-1
报数后状态: 1    0    0    4
报数游戏最后剩下的人是: 1
```

图 15 程序执行过程中 `start-1` 的状态变化情况

显然，`ring[-1]` 导致数组越界访问，造成堆内存破坏也就不足为奇了。

(5) 找到问题所在，编辑修正代码。

问题是找到了，如何修改。其实，在上文中已经讨论了 `start++` 导致超出数组末尾的情况和解决方法：`start = start % n;`

现在的问题刚刚相反，`start-1` 越过数组开头的情况。处理方法类似：取余运算。

```
ring[(n+start-1)%n] = 0;
```

编辑修正代码，编译链接执行，程序输出正确无误且无堆内存破坏，大功告成。

3.2 使用断言技术

断言（`assertion`），是关于程序执行状态的一些预期假设，在程序执行过程中可以跟踪捕捉这些断言是否成立，如果成立则什么都不做，否则终止执行。

3.2.1 C++语言的 `assert` 宏

C++语言提供 `assert` 宏来实现断言技术。当程序在 `Debug` 调试模式下运行时，`assert` 宏可以判断给定假设表达式是否成立，若成立则继续执行程序，若不成立则打印错误提示并终止执行。`assert` 宏定义在 `cassert` 头文件中，原型为：

```
assert(假设表达式);
```

3.2.2 利用 `assert` 宏捕捉不符合预期的假设

针对报数游戏程序，可以使用 `assert` 宏判断 `start-1` 是否符合预期。例如：

```
#include<cassert> //assert 宏包含在头文件 cassert 中
using namespace std;
void playRound(int* ring, int n, int m, int& start) {
    while(m>0) {
```

```

        if(ring[start++]!=0) m--;
        start = start % n;//这种处理方式更简洁
    }
    assert(start-1>=0 && start-1<n); //判断数组下标>=0 且<n
    ring[start-1] = 0;
}

```

编译链接执行，输入 $n=4$ ， $m=3$ ，程序执行退出且在屏幕上输出如下结果：

```

请输入游戏的人数 n (n>=2): 4
请输入报数步距 m (m>=1): 3
Assertion failed:start-1>=0&&start-1<n, file Josephus.cpp,line 78

```

根据 `Assertion failed` 的输出结果可以看出，断言假设的表达式 “`start-1>=0 && start-1<n`” 在这里判断不成立，即出现数组下标 `start-1` 不在 $[0 \sim n-1]$ 的范围内的情况，出错代码在 `Josephus.cpp` 文件的第 78 行的位置。

既然数组下标 `start-1` 不在 $[0 \sim n-1]$ 的范围内，数组越界访问就会导致堆内存破坏。进一步分析推断，不难找到出错的根源进而修正代码。

断言技术对于程序编写意义非凡。执行程序是对变量或者对象进行加工处理，使其状态发生一系列改变并最终达到预期状态的过程。在编写程序的时候，对于每一行程序代码将会导致数据状态发生什么样的变化，我们通常是有比较明确的预期的。

针对这些预期，在编写程序的过程中我们可以大量植入 `assert` 断言，跟踪捕捉程序运行时这些预期是否能够得到满足，提高程序的健壮性。例如：

- (1) 检查某段代码执行之前必须具备的条件是否达到。
- (2) 检查某段代码执行之后变量或对象是否达到预期状态。

3.2.3 禁用 `assert` 宏

但是，在程序编写完成对外发布的时候，代码中的大量断言就失去了存在的价值，而且会影响程序执行的效率。此时编辑代码删除 `assert` 费时费事。

可以通过定义 `NDEBUG` 宏来禁用 `assert` 宏，使 `assert` 宏失去作用。禁用的方法是在 “`#include<cassert>`” 之前插入 “`#defineNDEBUG`” 宏定义代码。

```

#define NDEBUG
#include <cassert>
using namespace std;

```

3.3 使用调试工具

在 Visual Studio 集成开发环境中提供了功能强大的调试器 (Debugger)，可以自由地观察程序运行时的行为，跟踪代码逻辑错误的位置。主要技术手段包括：

- (1) 通过设置“断点”，中断程序执行；
- (2) 通过“数据提示”、“变量窗口”、“调用堆栈窗口”等工具查看数据状态；
- (3) 通过“逐语句”和“逐过程”实现单步执行程序。

以下就以报数游戏为例，阐述使用 Visual Studio 2010 调试器的基本方法。

3.3.1 设置断点 (F9)

断点 (breakpoint)，是我们期待程序执行中断的代码行，设置断点就是告诉调试器我们想让程序在断点所在的代码行暂停执行。

在 Visual Studio 2010 中设置断点，有三种操作方法：

- (1) 把光标放到想要中断执行的代码行，点击【调试/切换断点】菜单项；
- (2) 把光标放到想要中断执行的代码行，在键盘上按 F9 键；
- (3) 把鼠标移到想要中断执行的代码行，在其左侧的灰色边栏上双击。

此时，在左侧灰色边栏上出现了一个红色的圆点，表示断点设置成功（图 16）。在同一代码行按上述操作方法反复操作，可以像按开关一样设置和删除断点。

对于报数游戏，通过函数调用关系分析，基本可以确定 BUG 隐藏在 playRound 函数中，需要观察 playRound 函数的执行过程，故需要设置断点中断 playRound 函数的执行。假设我们设置两个断点，一个在函数开始处，一个在函数结尾处。

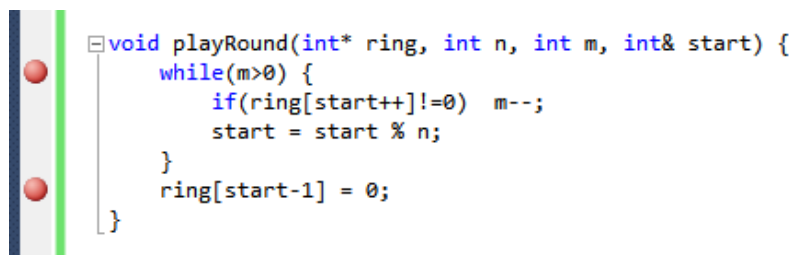


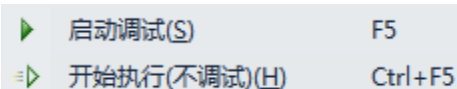
图 16 在 playRound 函数第一行代码处设置断点

3.3.2 启动调试 (F5)

断点设置完成，下一步是执行程序并且在执行到断点的时候中断程序执行。

在 Visual Studio 中提供了两种程序执行模式：调试执行和不调试执行，分别对应于【调试/启动调试】和【调试/开始执行（不调试）】两个菜单项。

只有在调试执行模式下，设置的断点才会被启用，执行遇到断点会中断；在不调试执行模式下，断点无效，不会导致中断执行。因此，要对报数游戏程序进行调试，需要点击【调试/启动调试】菜单项，或者在键盘上按 F5 启动调试执行。



此时，报数游戏程序开始执行。

(1) 首先跳出输入界面，等待输入：输入 n=3, m=4。

(2) 然后，执行流程遇到第一个断点，中断程序执行。此时，在断点的红色圆点里面出现一个黄色箭头，表示程序执行暂停在这一行（图 17）。

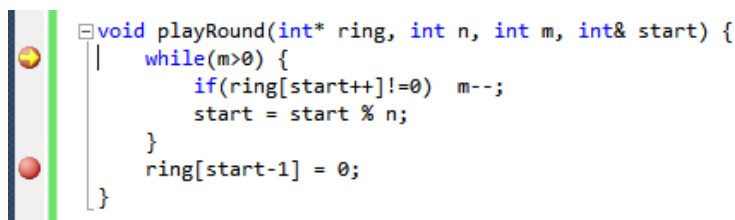


图 17 在 playRound 函数第一行代码的断点处中断执行程序

3.3.3 查看数据状态

程序中断执行后可以借助调试器的查看工具，查看当前的数据状态。Visual Studio 2010 提供了诸如数据提示窗口、变量窗口、调用堆栈窗口等数据状态查看工具。

(1) 数据提示窗口 (DataTip)

“数据提示”窗口，是最方便的查看数据状态的工具。在中断执行后将鼠标指针移到代码窗口的变量上面稍等一下，就会出现“数据提示”窗口。

例如，把光标移到 playRound 函数的参数 m 上面，出现图 18 所示的数据提示窗口，显示参数 m 的当前值为 3，请读者自行尝试熟练掌握该工具。

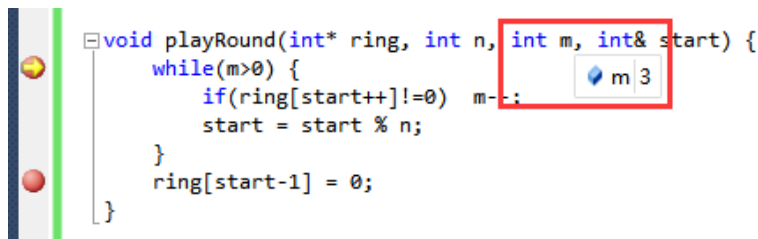


图 18 通过数据提示窗口查看参数 m 的状态

(2) 变量窗口

调试器提供了多种类型的变量窗口，用于方便地查看数据状态。在【调试/窗口】菜单下，点击【监视(W)】、【局部窗口(L)】、【自动窗口(A)】、【即时(I)】菜单项，可以打开这些查看窗口，具体窗口的外观参见图 19～图 22。

a) 局部变量窗口：显示当前函数中所有局部变量的状态（图 19）。

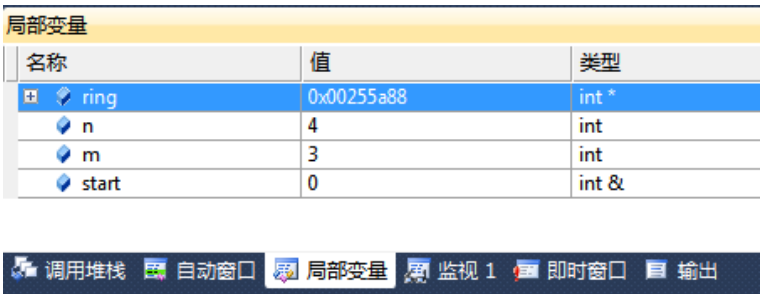


图 19 通过局部变量窗口查看数据状态

b) 自动窗口：显示当前代码和上一行代码中使用的所有变量的状态（图 20）。

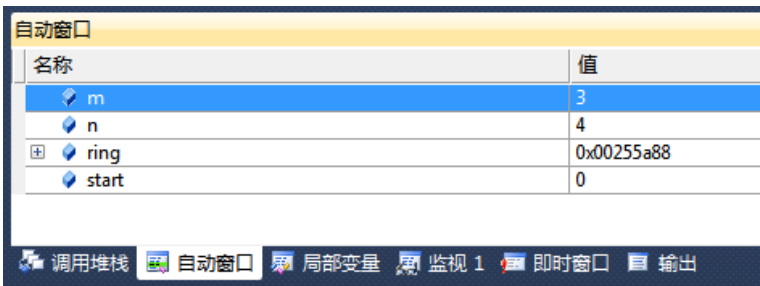


图 20 通过自动窗口查看数据状态

c) 监视窗口：添加要监视其值的变量，显示监视变量的状态。图 21 中，我们向监视窗口中添加了 ring[0]，其值为 1。可以添加更多的监视变量。

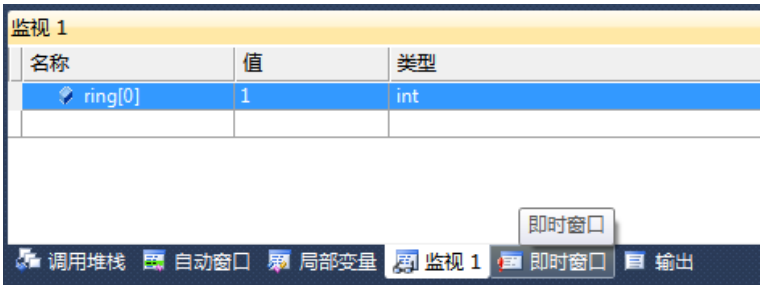


图 21 通过监视窗口查看数据状态

d) 即时窗口：即时计算并显示表达式的求值结果。例如，在图 22 所示的即时窗口中，键入 `m` 并回车，显示 `m` 的值为 3，键入 `n` 回车，显示 `n` 的值为 4。

对于 `playRound` 函数想了解 `ring` 数组当前的数据状态，可以在即时窗口中输入 `ring[0]` 回车，……，`ring[3]` 回车，依次查看 `ring` 数组的每个元素。



图 22 通过即时窗口查看数据状态

(3) 调用堆栈窗口

在中断模式下可以借助调用堆栈窗口，查看当前时刻的函数调用关系。在【调试/窗口】菜单下，点击【调用堆栈(C)】菜单项即可打开调用堆栈窗口。

例如，图 23 就是 `playRound` 函数中断执行时的函数调用堆栈。注意：当前调用的函数在最上面，下面的函数依次调用其上面的函数。可以看出，当前调用的函数是 `playRound`；`playRound` 函数被 `play` 函数调用；`play` 函数被 `main` 函数调用。

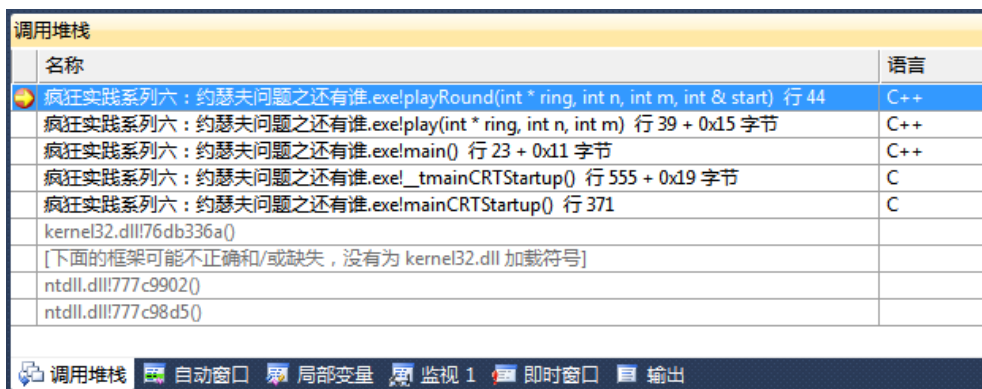


图 23 通过调用堆栈窗口查看当前的函数调用关系

通过调用堆栈可以了解当前函数调用的来龙去脉，尤其是当程序执行没有征兆的突然崩溃，此时就可以查看调用堆栈，快速定位到出错代码的位置。

3.3.4 程序单步执行

程序执行在断点处中断，可以使用“逐语句”、“逐过程”、“跳出”等工具进行程序的单步执行，即每次执行一行代码并暂停。其中：

（1）逐语句：一行一行执行代码，当遇到调用函数的语句时则进入被调函数中，在被调函数的第一行代码处暂停。“逐语句”的操作方法是选择【调试/逐语句】菜单项或者在键盘上按 F11 键。

（2）逐过程：一行一行执行代码，但是遇到调用函数的语句，不会进入到被调函数中，而是把函数调用当初一行代码执行，在函数调用语句的下一行暂停。“逐过程”的操作方法是选择【调试/逐过程】菜单项或者在键盘上按 F10 键。

（3）跳出：用于从一个函数内部跳出，返回到函数调用点。“跳出”的操作方法是选择【调试/跳出】菜单项或者在键盘上按 Shift+F11 键。

以图 17 所示的 playRound 函数调试为例，每按一次 F10 键“逐过程”执行，最初位于红色断点中的黄色箭头会随着执行流程移动一个代码行，表示单步执行了上一行代码。此时，随着变量状态的改变，相应的局部变量、自动窗口、监视窗口中内容也会随之改变，从而可以观察分析数据状态是否符合预期，如图 24 所示。

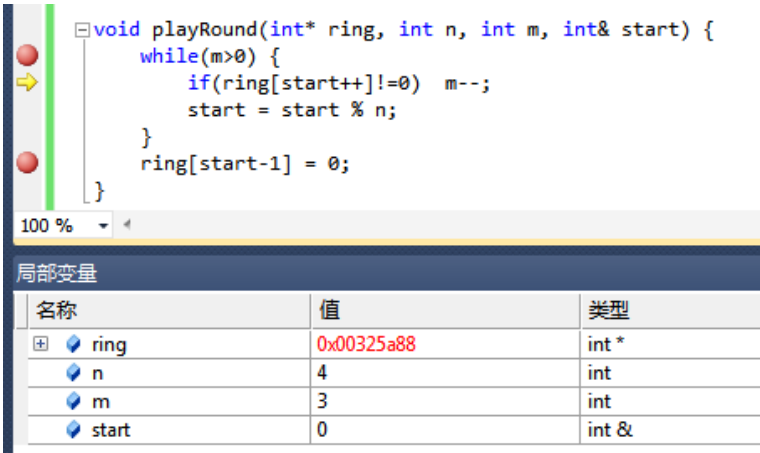


图 24 playRound 函数单步执行一次

3.3.5 加快程序调试 (F5)

如果感觉单步执行太慢，可以选择【调试/继续】或者按 F5 键，该操作会导致程序持续执行，直到遇到一个断点时暂停，从而加快程序调试的进程（图 25）。

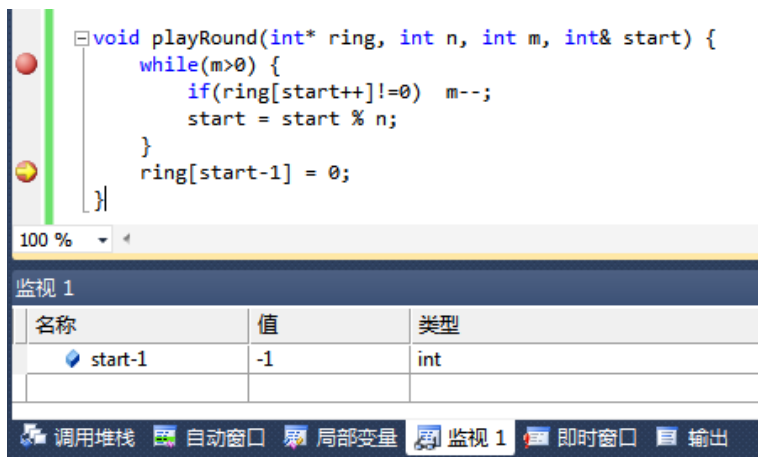


图 25 playRound 函数 F5 继续执行多次加快调试

对于 playRound 函数的调试，首先在监视窗口中增加监视表达式 start-1，用于监视数组下标是否符合预期；然后按 F5 继续执行，观察 start-1 的状态变化；多次 F5 之后，可以观察到数组下标 start-1 的取值变成 -1，数组越界即问题所在。

4. 更高效的问题求解思路

通过扫描数组模拟报数求解约瑟夫问题比较笨拙。但是，笨拙的问题求解思路，往往会对我们的 C++ 语言编程能力提出挑战，考验我们的语言功底。当然，如果我们深入分析一下，约瑟夫问题还是可以发现一些更为高效的求解方法。

对于从 0 到 n-1 的数列，假设 k 因为报数到 3 而出局，那么，我们就可以从 k+1 开始重新构造出一个从 0 到 n-2 的数列，构造方式如下所示：

0	1	...	k-1	k	k+1	k+2	...	n-1
↓	↓		↓		↓	↓		↓
n-k	n-k+1	...	n-2	/	0	1	...	n-k-1

不难看出，我们可以在构造出的新数列 0~(n-2) 上继续报数，实际上，问题已经演变成 n-1 人的报数问题。我们假设该 n-1 人报数问题的解，即最后剩余者的编号为 p(n-1)，假设 n 人报数问题的解记做 p(n)，则根据上面的数列构造方式可知：

(1) 如果 $p(n-1) = 0 \Rightarrow p(n) = k+1$;

(2) 如果 $p(n-1) = 1 \Rightarrow p(n) = k+2$;

(3) 如果 $p(n-1) = 2 \Rightarrow p(n) = k+3$;

.....

(4) 如果 $p(n-1) = n-2 \Rightarrow p(n) = k-1$ 。

分析总结其中的规律，可以得出递推公式： $p(n) = (p(n-1) + m) \% n$ 。

对于本文的问题， $m=3$ ，而且对于 1 人报数问题： $p(1) = 0$ 。进而可知，

$$p(2) = (p(1) + 3) \% 2 = 1$$

$$p(3) = (p(2) + 3) \% 3 = 1$$

$$p(4) = (p(3) + 3) \% 4 = 0$$

继续递推过程即可求得 $p(n)$ 的值。递推思路的约瑟夫问题的代码如下：

```
int main() {
    int n, m; cin >> n >> m;
    for(int p1=0, i=2; i<=n; i++) p1=(p1+m)%i;
    cout<<"最后剩下的是: "<<p1+1<<"号"<<endl;
    return 0;
}
```

这种解法比模拟游戏过程的解法要高效的多，编程实现也更容易一些。但是，问题求解思路却更为复杂，理解起来也要困难一些。

疯狂实践系列七：螺旋输出 $1 \sim N^2$

题目：输入整数 N ，先按升序将 $1 \sim N*N$ 的数输出成 N 行 N 列的方形，然后再从 1 开始按从外向内顺时针螺旋的顺序依次输出每个整数。假设 $N=4$ ，输出如下：

4 行 4 列输出结果：

```
1  2  3  4
5  6  7  8
9 10 11 12
13 14 15 16
```

顺时针螺旋输出的结果：

```
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

1. 训练目标

- (1) 理解自顶向下、分而治之的问题求解思维。
- (2) 体会编程并非一蹴而就，需要不断自省，反复迭代修正。
- (3) 理解并掌握二维及多维动态数组的创建和释放方法。
- (4) 条条大路通罗马，换一种思路可能会产生更优雅的程序。

2. 自顶向下分治

2.1 第一层问题：main 函数

编写 main 函数，处理输入输出，假设存在函数 printSquare 和 printSpiral 能实现 $N*N$ 方形输出和顺时针螺旋输出

```
int main() {
    int N;
    cout<<"请输入 N: ";
    cin >> N;
    cout<<N<<"行"<<N<<"列输出结果: "<<endl;
    printSquare(N); // N*N 方形输出
    cout<<"顺时针螺旋输出的结果: "<<endl;
    printSpiral(N); // 顺时针螺旋输出
```

```
    return 0;
}
```

现在问题分解归结为定义 `printSquare` 函数和 `printSpiral` 函数。其中，

(1) 函数 `printSquare`: 用于实现 $N*N$ 方形输出。其函数原型为:

```
void printSquare(int);
```

(2) 函数 `printSpiral`: 用于实现顺时针螺旋输出。其函数原型为:

```
void printSpiral(int);
```

2.2 第二层问题：定义 `printSquare` 函数

函数 `printSquare`, 按升序将 $1 \sim N*N$ 的数输出成 N 行 N 列的方形, 实际上就是在按升序顺序输出过程中增加了几个回车输出。代码如下:

```
void printSquare(int n){
    for(int i=1; i<=n*n; i++) {
        cout<<i<<"\t";
        if(0 == i%n) cout<<endl; //n 的倍数时换行
    }
    cout<<endl;
}
```

2.3 第二层问题：定义 `printSpiral` 函数

函数 `printSpiral`, 将 $1 \sim N*N$ 按顺时针螺旋输出。解决思路有很多, 这里把整个螺旋输出问题分解为每次输出一圈的问题。尝试编写代码如下:

```
void printSpiral(int n){
    int rounds = getRounds(n); //getRounds 计算需要输出几圈
    for(int i=0; i<rounds; i++) {
        printRound(i); //printRound 输出第 i 圈
    }
    cout<<endl;
}
```


问题归结为定义 `getRounds` 函数和 `printRound` 函数。其中，`getRounds` 用于计算需要输出几圈，`printRound` 用于输出第 `i` 圈。假设二者的函数原型分别为：

```
int getRounds(int n);
void printRound(int i); //参数暂定只有一个 i
```

2.3.1 定义 `getRounds` 函数

`getRounds` 函数用于计算需要输出几圈。分析可得：如果 `N` 是偶数，圈数应为 $N/2$ ，如果 `N` 为奇数，圈数应为 $N/2+1$ 。实现代码如下：

```
int getRounds(int n) { return n%2==0 ? n/2 : n/2+1; }
```

`getRounds` 函数非常小，推荐将函数定义为 `inline` 函数。

2.3.2 定义 `printRound` 函数

`printRound` 用于输出第 `i` 圈，故可以推断出函数至少要有一个参数 `i`。问题求解思路同样简单，把输出一圈的问题分解为输出四个边的问题。尝试编写如下：

```
void printRound(int i) {
    int start = getStart(i); //计算第 i 圈输出起点 start
    int length = getLength(i); //计算第 i 圈输出边长 length,
    printRight(start, length); //向右输出
    printDown(start, length-1, n); //向下输出
    printLeft(start, length-1, n); //向左输出
    printUp(start, length-2, n); //向上输出
}
```

基本思路是先计算第 `i` 圈输出的起点 `start`，然后计算第 `i` 圈输出的边长 `length`，然后依次向右输出 `length` 个数，步距加 1；向下输出 `length-1` 个数，步距加 `n`；向左输出 `length-1` 个数，步距减 1；向上输出 `length-2` 个数，步距减 `n`。

在编写程序过程中，发现 `printDown` 和 `printUp` 函数需要设定输出步距 `n`，因此在 `printRound` 函数中增加一个输出步距参数。修改 `printRound` 函数代码如下：

```
void printRound(int i, int n) {……}
```

`printRound` 函数增加一个参数，其函数调用代码中也就需要增加一个实参。在函数 `printSpiral` 中调用了 `printRound` 函数，请修改该处代码。

程序编写从来就不是一帆风顺的事情，初始只能做一些宏观概况性的假设和推断。随着程序编写的深入，这些假设和推断不断得到澄清和确认，此时就需要程序员不断地回头对已有代码进行修正，不断地迭代精化，逼近期望的完美程序。

至此，通过层层分治，我们将问题归结为定义 `getStart`、`getLength`、`printRight`、`printDown`、`printLeft`、`printUp` 六个函数，然后徐徐图之。

(1) 定义 `getStart` 函数

函数 `getStart` 计算第 `i` 圈输出起点 `start`。分析推断可得：第 `i` 圈的起点是位于 `i` 行 `i` 列的元素，根据每行的元素个数 `N` 可以计算得出。代码如下：

```
int getStart(int i, int n) { return i*n+i+1; }
```

显然，在 `printRound` 函数定义中 `getStart` 函数的调用代码缺失了第二个参数，此时需要回过头来修改 `printRound` 函数的定义，请读者自行完成。

(2) 定义 `getLength` 函数

函数 `getLength` 计算第 `i` 圈输出的边长有多少个元素，分析推断可得：每输出一圈输出的边长元素个数就减去 2，代码实现如下：

```
int getLength(int i, int n) { return n-2*i; }
```

同样，在 `printRound` 函数定义中 `getLength` 函数的调用代码缺失了第二个参数，需要回过头来修改 `printRound` 函数定义，请读者自行完成。

(3) 定义 `printRight` 函数

函数 `printRight`，用于从 `start` 开始向右输出 `length` 个数并更新 `start`。

```
void printRight(int &start, int length) {  
    for(int i=0; i<length; i++) cout<<start++<<" ";  
    start--;  
}
```

在 `printRight` 函数输出过程中，每次输出都要同时将 `start` 加 1，使其保持为当前要输出的元素。最终 `printRight` 函数输出完成之后，`start` 可以用于 `printDown` 函数的输出起点。因此，将 `start` 参数定义为引用类型。

(4) 定义 `printDown` 函数

函数 `printDown`，用于从 `start` 开始向下输出 `length` 个数并更新 `start`，每次输出的步距为 `n`。代码实现如下：

```
void printDown(int &start, int length, int n) {
    for(int i=0; i<length; i++) cout<<(start+=n)<<" ";
}
```

(5) 定义 printLeft 函数

函数 printLeft, 用于从 start 开始向左输出 length 个数并更新 start。

```
void printLeft(int &start, int length) {
    for(int i=0; i<length; i++) cout<<(start-=1)<<" ";
}
```

(6) 定义 printUp 函数

函数 printUp, 用于从 start 开始向上输出 length 个数, 每次输出的步距为 n。代码实现如下:

```
void printUp(int &start, int length, int n) {
    for(int i=0; i<length; i++) cout<<(start-=n)<<" ";
}
```

3. 换个思路：遍历地图

换一种新的思路, 尝试借助二维数组进行问题求解。把 1~N*N 放到如下一个二维数组中, 0 表示围墙不能通行, 不妨称之为**地图** (如图 26 所示)。然后从 (1, 1) 这个位置开始沿着围墙顺时针访问地图每个方格, 每访问一个方格, 就将该方格的内容置为 0, 也就是说, 将该方格设置成围墙。遍历结束所有方格都变成围墙。

0	0	0	0	0	0
0	1	2	3	4	0
0	5	6	7	8	0
0	9	10	11	12	0
0	13	14	15	16	0
0	0	0	0	0	0

图 26 螺旋输出的二维数组遍历地图

3.1 重新定义 printSpiral 函数

根据上述思路尝试重新编写函数 printSpiral, 代码如下:

```
void printSpiral(int n){
```

```
int **map = createMap(n); //创建遍历地图
walk(map, n); //从(1,1)位置开始遍历地图
deleteMap(map, n); //释放遍历地图
}
```

在 `printSpiral` 函数中调用了其他三个函数，代码非常简单，想要出错都不可能。

问题归结为定义三个函数 `createMap`、`walk`、`deleteMap`。其中，`createMap` 创建地图，`walk` 遍历地图，`deleteMap` 释放地图。函数原型如下：

```
int** createMap(int n);
void deleteMap(int **map, int n)
void walk(int **map, int n)
```

3.1.1 创建地图：定义 `createMap` 函数

`createMap` 函数创建遍历的地图，需要根据输入的整数 `N` 动态创建一个二维数组，然后将二维数组初始化为图 26 所示的内容。代码如下：

```
int ** createMap(int n) {
    int **map = new int*[n+2];
    for(int i=0; i<n+2; i++) {
        map[i] = new int[n+2];
        for(int j=0; j<n+2; j++) map[i][j] = 0;
    }
    int start = 1;
    for(int i=1; i<n+1; i++)
        for(int j=1; j<n+1; j++) map[i][j] = start++;
    return map;
}
```

3.1.2 释放地图：定义 `deleteMap` 函数

`createMap` 函数通过 `new` 动态创建地图的二维数组，必须 `delete`。`deleteMap` 函数专门用于释放地图对应的二维数组。代码如下：

```
void deleteMap(int **map, int n) {
    for(int i=0; i<n+2; i++) delete[] map[i];
}
```

```
delete[] map;
}
```

3.1.3 遍历地图：定义 walk 函数

walk 函数用于遍历地图。从地图的(1, 1)位置开始，输出当前位置的内容并将当前位置的内容置 0，然后将当前位置更新为沿围墙顺时针移动的下一个位置。反复执行这一过程，直到沿围墙顺时针移动找不到非围墙的位置。

这里假设存在 moveNext 函数，可以更新当前位置(i, j)。如果能找到下一个位置，则更新(i, j)且返回 true；如果找不到下一个位置，返回 false。

利用 moveNext 函数，可以比较轻松地编写出 walk 函数，代码如下：

```
void walk(int **map, int n){
    int i = 1, j = 1;
    while(true) {
        cout<<map[i][j]<<" ";
        map[i][j] = 0; //将位置(i,j)设置为围墙
        if(!moveNext(map, i, j)) break; //找不到下一个位置退出循环
    }
}
```

3.1.4 计算下一个位置：定义 moveNext 函数

给定当前位置(i, j)，moveNext 函数用于计算沿围墙顺时针移动的下一个位置，如果找到则用该位置更新当前位置(i, j)并返回 true，如果找不到则返回 false。

由于要通过形参修改实参，故(i, j)两个参数定义为引用类型。

```
bool moveNext(int **map, int &i, int &j){//沿着围墙移动
    if(map[i-1][j]==0 && map[i][j+1]!=0) { j++; return true;}
    if(map[i][j+1]==0 && map[i+1][j]!=0) { i++; return true;}
    if(map[i+1][j]==0 && map[i][j-1]!=0) { j--; return true;}
    if(map[i][j-1]==0 && map[i-1][j]!=0) { i--; return true;}
    return false;
}
```

这里，`map[i-1][j]==0` 表示当前位置的上方是围墙，`map[i][j+1]!=0` 表示当前位置的右边不是围墙。因此，`moveNext` 函数中 5 行代码的含义分别是：

第 1 行代码：如果当前位置的上方是围墙且右边非围墙，则向右移动（`j++`）。

第 2 行代码：如果当前位置的右边是围墙且下方非围墙，则向下移动（`i++`）；

第 3 行代码：如果当前位置的下方是围墙且左边非围墙，则向左移动（`j--`）；

第 4 行代码：如果当前位置的左边是围墙且上方非围墙，则向上移动（`i--`）；

第 5 行代码：否则找不到下一个位置，返回 `false`。

3.2 条条大路通罗马

条条大路通罗马，换一种问题求解思路，或许可以编写出更优雅的程序。对于语言学习者而言，重点是通过大量的实践训练实现 C++ 语言的融合内化，把计算机语言变成我们自己本能的母语，如臂使指，灵活自如地表达问题求解思路。

把复杂问题通过自顶向下分治，归结为大量的细小问题，每个细小问题的函数实现很小，代码逻辑非常简单，出错可能性很小，自然而然程序更健壮。

无论一种问题求解思路是笨拙还是精巧，语言学习者都要有能力清晰顺畅地将其表达出来。至于完美的代码呈现并不重要，请读者根据上文自行整理。

疯狂实践系列八：排座座吃果果

题目：随机产生 N 个小于 1000 的数写入文件 `todo.dat`，对文件 `todo.dat` 中的数据按从小到大排序后写入文件 `done.dat`。

1. 训练目标

- (1) 理解自顶向下、分而治之的问题求解思维。
- (2) 掌握使用文件输入输出流进行文件读写的方法。
- (3) 掌握数组、动态数组及数组用作函数参数的语法。
- (4) 掌握函数指针及利用函数指针实现通用函数的方法。

2. 自顶向下分治

2.1 第一层问题：main 函数

根据自顶向下的分治思维，先编写最顶层的 `main` 函数，完成最基本的数据准备、数据输入输出、排序处理等工作。其中，假设存在如下 5 个函数：

- (1) `createData` 函数：用于产生数据写入文件。其函数原型为：

```
void createData(string, int);
```

- (2) `readData` 函数：用于从文件读取数据。其函数原型为：

```
void readData(double*, int, string);
```

- (3) `writeData` 函数：用于将排序结果写入文件。其函数原型为：

```
void writeData(double*, int, string);
```

- (4) `xsort` 函数：用于实现排序。其函数原型为：

```
void xsort(double*, int);
```

- (5) `print` 函数：用于输出数据。其函数原型为：

```
void print(double*, int);
```

如果上述 5 个函数存在，则 `main` 函数的编写非常容易。代码如下：

```
const int MAXINT = 1000;
int main() {
```

```
const int N = 10;
string todoFile = "todo.dat";
string doneFile = "done.dat";
createData(todoFile, N); //产生数据写入 todo.dat
double *nums = new double[N]; //创建动态数组
readData(nums, N, todoFile); //读出数据到 nums 数组
cout<<"原始数据: "; print(nums, N); //输出原始数据
xsort(nums, N); //排序处理
cout<<"排序数据: "; print(nums, N); //输出排序结果
writeData(nums, N, doneFile);
delete[] nums; //释放动态数组
return 0;
}
```

问题归结为定义 createData、readData、writeData、xsort、print 五个函数。

2.2 第二层问题：数据的产生和读写

用于产生数据和读写数据的 createData、readData、writeData 三个函数比较简单，主要是掌握使用文件输入输出流读写文件的方法。代码编写如下：

```
void createData(string fname, int n){
    srand(time(0)); //随机种子
    ofstream ofs(fname); //文件输出流
    for(int i=0; i<n; i++) { ofs<<rand()%MAXINT<<endl; }
    ofs.close();
}

void readData(double* nums, int n, string fname) {
    ifstream ifs(fname); //文件输入流
    for(int i=0; i<n && !ifs.eof(); i++) { ifs>>nums[i]; }
    ifs.close();
}

void writeData(double* nums, int n, string fname) {
```



```

ofstream ofs(fname); //文件输出流
for(int i=0; i<n; i++) { ofs<<nums[i]<<endl; }
ofs.close();
}

```

用于数据输出的 `print` 函数同样非常简单，代码编写如下：

```

void print(double* nums, int n) {
    for(int i=0; i<n; i++) cout<<nums[i]<<" ";
    cout<<endl;
}

```

2.3 第二层问题：选择排序及 `xsort` 函数

排序是计算机科学领域中的经典问题，存在很多经典的解决思路，如冒泡排序、选择排序、插入排序、快速排序、堆排序、归并排序等算法。

2.3.1 选择排序的基本思路

这里采用了最简单直观的选择排序思路。例如，对于如下的 `nums` 数组，

```
double nums[] = {-25, 32, -41, 5, -78, 44, 31};
```

数组初始状态如图所示：

下标	0	1	2	3	4	5	6
nums	-25	32	-41	5	-78	44	31

假设我们要按数值从小到大进行排序，则具体排序过程演示如下：

- (1) 找出下标 0~6 的元素中的最大元素，将其与数组第 6 个元素交换；

下标	0	1	2	3	4	5	6
nums	-25	32	-41	5	-78	31	44

- (2) 找出下标 0~5 的元素中的最大元素，将其与数组第 5 个元素交换；

下标	0	1	2	3	4	5	6
nums	-25	31	-41	5	-78	32	44

- (3) 找出下标 0~4 的元素中的最大元素，将其与数组第 4 个元素交换；

下标	0	1	2	3	4	5	6
nums	-25	-78	-41	5	31	32	44

- (4) 重复这一过程，……

- (5) 找出下标 0~1 的元素中的最大元素，将其与数组第 1 个元素交换；

下标	0	1	2	3	4	5	6
nums	-78	-41	-25	5	31	32	44

至此，在 `nums` 中就得到了按原始值从小到大排序的一组数值。这种排序的思路简单而直观，称为**选择排序法**。

2.3.2 选择排序函数 `xsort`

函数 `xsort` 实现选择排序。假设待排序数组的元素个数为 `n`，同时将循环变量 `i` 初始设置为 `N-1`。则在 `xsort` 函数中，重复执行 `n-1` 次如下操作：

- (1) 从数组的 `0~i` 个元素中找出最大元素所在的位置；
- (2) 将最大元素与第 `i` 个元素进行交换。

假设 `findMax` 函数可以找到最大元素位置，`swap` 函数可以实现交换，则排序函数 `xsort` 实现起来非常容易。代码编写如下：

```
void xsort(double* nums, int N){
    for (int i=N-1 ; i>0 ; i--) {
        int pos = findMax(nums, i+1); //找出最大元素的位置
        swap(nums, i, pos); //将最大元素与 nums[i]交换
    }
}
```

问题被分治归结为更简单的两个问题：找最大元素的位置和交换。其中，`findMax` 用于找最大元素，`swap` 用于实现交换。函数原型为：

```
int findMax(double*, int);
void swap(double*, int, int); //swap 实现交换
```

2.3.3 `findMax` 函数

`findMax` 函数用于寻找最大元素的位置，实现思维非常简单：假设最大元素所在的位置为 `imax`，初始值为 `0`；然后将 `nums[imax]` 依次与数组其他元素比较，如果遇到更大的元素，就将 `imax` 修改为该数组元素的位置。代码如下：

```
int findMax(double* nums, int m) {
    int imax = 0;
    for (int i=1; i<m; i++)
        if (nums[i]>nums[imax]) imax = i;
```

```
    return imax;
}
```

2.3.4 swap 函数

swap 函数，实现两个数组元素的交换。代码如下：

```
void swap(double* nums, int i, int j) { //swap 实现交换
    double tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
}
```

3. 实现从大到小排序

把问题稍作变换，要求从大到小排序，该当如何呢？有了前面的基础，这个问题的解决思路就很简单了：在选择排序过程中每次找到最小元素，将其与最后一个元素做交换，经过 $N-1$ 次操作最终形成从大到小的排序结果。

假设存在 findMin 函数可以找数组最小元素，函数原型为：

```
int findMin(double*, int);
```

则定义从大到小的排序函数 nsort，易如反掌。代码编写如下：

```
void nsort(double* nums, int N){
    for (int i=N-1 ; i>0 ; i--) {
        int pos = findMin(nums, i+1);
        swap(nums, i, pos);
    }
}

int findMin(double* nums, int m) {
    int imin = 0;
    for (int i=1; i<m; i++)
        if (nums[i]<nums[imin]) imin = i;
    return imin;
}
```

4. 通用排序函数 gsort

观察 `xsort` 函数和 `nsort` 函数，不难发现二者长得非常相像，区别只是 `xsort` 调用 `findMax`，而 `nsort` 调用 `findMin`。能否把两个函数合成一个？

借助 C++ 语言的函数指针，可以把 `findMax` 和 `findMin` 处理成待定的函数形参，形参类型为函数指针，从而实现一个通用排序函数 `gsort`。代码如下：

```
typedef int (*FINDELEM)(double*, int); //函数指针类型
void gsort(double* nums, int N, FINDELEM find){
    for (int i=N-1 ; i>0 ; i--) {
        int pos = find(nums, i+1);
        swap(nums, i, pos);
    }
}
```

利用 `gsort` 通用排序函数，可以轻松地从小到大或者从大到小对数组进行排序。

```
double nums[] = {-25, 32, -41, 5, -78, 44, 31};
int nsize = sizeof(nums)/sizeof(double);
gsort(nums, nsize, findMax); //按原始值从小到大排序
gsort(nums, nsize, findMin); //按原始值从大到小排序
```

读者如果仔细观察还会发现，`findMax` 函数和 `findMin` 函数长得也很像，请你尝试使用函数指针将 `findMax` 和 `findMin` 合并成一个通用函数。

5. 完整代码组织结构

```
//源文件: main.cpp
#include "stdafx.h"
#include "prepdata.h"
#include "sort.h"
#include <iostream>
using namespace std;
int main() {
    const int N = 10;
    string todoFile = "todo.dat";
    string doneFile = "done.dat";
```

```

    createData(todoFile, N);
    double *nums = new double[N];
    readData(nums, N, todoFile);
    cout<<"原始数据: "; print(nums, N);
    gsort(nums, N, findMax);
    cout<<"排序数据: "; print(nums, N);
    writeData(nums, N, doneFile);
    delete[] nums;
    return 0;
}

```

//头文件: sort.h

```

void print(double*, int);
int findMax(double*, int);
int findMin(double*, int);
void swap(double*, int, int);
typedef int (*FINDELEM)(double*, int);
void gsort(double*, int, FINDELEM);

```

//源文件: sort.cpp

```

#include "stdafx.h"
#include "sort.h"
void gsort(double* nums, int N, FINDELEM find) {...}
int findMin(double* nums, int m) {...}
int findMax(double* nums, int m) {...}
void swap(double* nums, int i, int j) {...}

```

//头文件: prepdata.h

```

#include<string>
usingnamespace std;
const int MAXINT = 1000;
void createData(string, int);
void readData(double*, int, string);
void writeData(double*, int, string);

```

//源文件: prepdata.cpp

```

#include "stdafx.h"
#include "prepdata.h"
#include <fstream>
#include <iostream>
#include <ctime>
using namespace std;
void createData(string fname, int n){...}

```

```
void writeData(double* nums, int n, string fname) {...}  
void readData(double* nums, int n, string fname) {...}  
void print(double* nums, int n) {...}
```

疯狂实践系列九：日期问题

题目：编写程序完成关于日期的系列处理操作。主要功能包括：

- (1) 计算两个日期之间的间隔天数。如 2018-11-25 和 2018-11-26 间隔 1 天；
- (2) 计算给定日期 n 天后的日期，如 2018-11-25 之后 1 天是 2018-11-26。

1. 训练目标

- (1) 理解自顶向下、分而治之的问题求解思维。
- (2) 掌握通过定义类描述对象属性和行为的方法和语法。
- (3) 掌握根据对象的使用场景定义类的构造函数的方法。
- (4) 根据对象运算需求，反演运算符重载函数原型的方法。
- (5) 掌握通过成员函数方式，进行运算符重载的语法。
- (6) 掌握通过非成员方式，重载输入输出运算符的语法。
- (7) 了解常用单元测试的方法，掌握 VS2010 的单元测试工具。

2. 自顶向下分治

本系列引入**面向对象问题求解思维**，问题求解思路非常简单，假设存在一个日期类 CDate，通过与 CDate 对象的沟通协作可以完成问题要求的计算。

2.1 第一层问题：main 函数

在 main 函数中需根据问题需求，对 CDate 类的属性和行为做出推断和假设：

- (1) 假设 CDate 对象能够进行输入输出运算>>和<<，实现日期输入输出；
- (2) 假设 CDate 对象能够进行减法运算得到间隔天数：date1-date2；
- (3) 假设 CDate 对象可以进行大小比较运算：date1 > date2；
- (4) 假设 CDate 对象可以进行加法运算得到 n 天后日期：date1 + n；
- (5) 假设 CDate 对象可以进行相等判断，date1 == date2。

有了 CDate 类及上述假设的行为能力，编写 main 函数易如反掌，代码如下：

```
int main() {
```

```

CDate date1, date2;
cin>>date1>>date2; //假设 CDate 具有输入运算>>
cout<<"第 1 个日期是: "<<date1<<endl; //假设 CDate 具有输出运算
cout<<"第 2 个日期是: "<<date2<<endl;
int span = date1 - date2; //假设 CDate 具有减法运算
cout<<"两个日期的间隔天数: "<<span<<endl;
int gt = date1 > date2; //假设 CDate 可以比较大小
//假设 CDate 具有加法运算和相等判断运算
if((gt>0&&date2+span==date1)|| (gt<0&&date1+span==date2))
    cout<<"经验证, 减法计算结果正确"<<endl;
return 0;
}

```

问题归结为 main 函数中关于 CDate 的系列假设, 定义 CDate 类使之成立。

2.2 第二层问题：定义 CDate 类

根据实际需求, CDate 日期类应该具有年、月、日三个属性和一系列行为能力, 如构造函数、普通成员函数、运算符重载函数 (>>、<<、-、>、+、==) 等。

2.2.1 CDate 类的构造函数

根据 CDate 类可能的使用场景, 可以推断 CDate 类应该具有如下构造函数:

- (1) 默认构造函数: 无参数, 没有提供任何初始状态信息; 例如,

```
CDate now; //调用默认构造函数, 默认日期是 1-1-1
```

- (2) 普通有参构造函数: 有参, 提供了一些初始状态信息; 例如,

```
CDate today(2008, 5, 1); //调用有参构造函数
```

- (3) 拷贝构造函数: 单参数, 提供了另一个同类对象; 例如,

```
CDate birthday = today; //调用拷贝构造函数, 或者
CDate birthday(today);
```

根据上述使用场景, 可以推断 CDate 类应该具有默认构造函数、普通三参数构造函数以, 无需提供拷贝构造函数, 由编译器自动合成。

2.2.2 CDate 类的运算符重载函数

根据 main 函数中的假设要求，CDate 类应该能够支持>>、<<、-、>、+、==等运算，需要提供相应的运算符重载函数。

C++语言重载操作符，可以采用采用成员或者非成员重载方式。对于任意二元操作符#，其运算表达式为 obj1 # obj2，等价的成员和非成员重载方式为：

(1) 成员函数调用形式：obj1.operator#(obj2)

(2) 非成员函数调用形式：operator#(obj1, obj2)

其中，成员重载方式要求操作符的左操作数必须是 CDate 类的对象。因此，运算符-、>、+、==可以采用成员重载方式，>>和<<只能采用非成员重载方式。考虑到>>和<<的非成员重载函数需要访问 CDate 类的私有数据成员，将其声明为友元函数。

2.2.3 定义 CDate 类的主体框架

根据 CDate 类的数据成员、构造函数、运算符重载函数分析，定义 CDate 如下：

```
class CDate{
public:
    //构造函数（包含默认构造函数）
    CDate(int y=1, int m=1, int d=1):year(y),month(m),day(d){}
    //通过成员函数实现操作符重载
    int operator-(const CDate&) const; //重载操作符-
    CDate operator+(int n); //重载操作符+
    int operator>(const CDate&); //重载操作符>
    bool operator==(const CDate&); //重载操作符>
    //通过非成员函数实现>>、<<操作符重载
    friend istream& operator>>(istream&, CDate&);
    friend ostream& operator<<(ostream&, const CDate&);
    //后续增加的其他辅助函数
    .....
private:
    int year, month, day;
```

```
};
```

2.2.4 实现 CDate 的输入输出

通过非成员函数重载操作符>>、<<，可以实现 CDate 对象的输入和输出。

(1) 输出运算符<<重载函数

```
ostream& operator<<(ostream& osm, const CDate& date) {
    osm<<date.year<<"年"<<date.month<<"月"<<date.day<<"日";
    return osm;
}
```

(2) 输入运算符>>重载函数

输入运算符的实现稍微复杂一些，因为月份和天数需进行合法性检查。

```
istream& operator>>(istream& ism, CDate& date) {
    date.inputYear();
    date.inputMonth();
    date.inputDay();
    return ism;
}
```

假设 CDate 类有三个辅助的成员函数：inputYear、inputMonth、inputDay。于是问题继续分解，归结为三个辅助成员函数的定义。

(3) 定义 inputYear、inputMonth、inputDay

inputYear 处理年份输入，inputMonth 处理月份输入，比较简单。代码如下：

```
void CDate::inputYear(){
    cout<<"请输入年份[1~]: "; cin>>year;
    while(year<1) { cout<<"年份错误请重输。"; cin>>year; }
}

void CDate::inputMonth(){
    cout<<"请输入月份[1-12]: "; cin>>month;
    while(month<1 || month>12) {
        cout<<"月份错误请重输。"; cin>>month;
    }
}
```

`inputDay` 处理天数输入，因闰年问题稍微复杂。代码如下：

```
void CDate::inputDay(){
    int maxDay = getDaysOfMonth(year, month); //某年某月的天数
    cout<<"请输入天数[1-<<maxDay<<]": "; cin>>day;
    while(day<1 || day>maxDay) {
        cout<<"天数错误请重输。";cin>>day;
    }
}
```

这里假设 `CDate` 类有成员函数 `getDaysOfMonth`，计算给定年份月份的天数。

```
static int CDate::getDaysOfMonth(int year, int month) {
    if(isLeapYear(year) && 2 == month) return 29; //闰 2 月
    return DAYS[month];
}
```

考虑到 `getDaysOfMonth` 函数，应该能够计算任意给定年份月份的天数，该成员函数不应依赖于特定 `CDate` 对象，故将该成员函数声明为 `static` 成员。

在 `getDaysOfMonth` 成员函数定义中有两个假设。其一，假设存在数组 `DAYS`，保存常规年份 12 个月每月的天数，故需定义常量数组 `DAYS`：

```
const int DAYS[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
```

其二，假设 `CDate` 类有成员函数 `isLeapYear`，判断是否闰年。基于同样的考虑，将 `isLeapYear` 函数声明为 `CDate` 类的静态成员。

```
static bool CDate::isLeapYear(int year){
    if(year%4==0&&year%100!=0||year%400==0) return true;
    return false;
}
```

2.2.5 实现 `CDate` 的比较运算 (>和==)

通过成员函数重载操作符>、==，可以实现 `CDate` 对象的大小比较和相等判断。

```
int CDate::operator>(const CDate& date){
    if (year > date.year) return 1;//年份大日期在后
    if (year < date.year) return -1;//年份小日期在前
```

```

    if (month > date.month) return 1; //年份相同月份大日期在后
    if (month < date.month) return -1; //年份相同月份小日期在前
    if (day > date.day) return 1; //年月相同天数大日期在后
    if (day < date.day) return -1; //年月相同天数小日期在前
    return 0; //相等
}

bool CDate::operator==(const CDate& date){
    return 0 == (*this>date);
}

```

2.2.6 实现 CDate 的减法运算

通过成员函数重载操作符`-`，可以实现 `CDate` 对象的减法运算，得到间隔天数。计算的思路比较简单，以 2006 年 10 月 17 日到 2018 年 9 月 29 日为例：

- (1) 从 2006 年开始到 2018-1 年，把每年的天数加起来；
- (2) 加上 2018 年的 1~8 月的天数，再加上 29；
- (3) 减去 2006 年 1~9 月的天数，再减去 17。

根据这一思路，`CDate` 类的减法运算符重载函数，定义如下：

```

int CDate::operator-(const CDate& date) const {
    int gt = (*this) > date;
    if(0 == gt) return 0; //两个日期相等，返回 0
    CDate d1(*this), d2(date); //d1 是较小日期，d2 是较大日期
    if(1 == gt) { d1=date; d2=*this; }
    int span = 0;
    for(int y=d1.year; y<d2.year; y++) { //(1)
        span += getDaysOfYear(year);
    }
    span += d2.getDaysOfDate(); //(2)
    span -= d1.getDaysOfDate(); //(3)
    return span;
}

```

这里，假设存在两个辅助函数 `getDaysOfYear` 和 `getDaysOfDate`。其中，

(1) `getDaysOfYear`，是静态成员函数，计算给定年份的总天数。

```
const int DaysOfLeapYear = 366;
const int DaysOfNonLeapYear = 365;
int CDate::getDaysOfYear(int year) {
    if(isLeapYear(year))    return DaysOfLeapYear;
    return DaysOfNonLeapYear;
}
```

(2) `getDaysOfDate`，是普通成员函数，计算本年从 1 月 1 号到今天的天数。

```
int CDate::getDaysOfDate() const {
    int days = 0;
    for(int m=1; m<month; m++) days+=getDaysOfMonth(year, m);
    days += day;
    return days;
}
```

2.2.7 实现 CDate 的加法运算

通过成员函数重载操作符`+`，可以实现 `CDate` 对象的加法运算，得到 `n` 天后的日期。计算的思路比较简单，以 2006 年 10 月 17 日加上 1000 天为例：

- (1) 直接将 1000 加到 `day` 数据成员上， $2006/10/17 + 1000 = 2006/10/1017$ 。
- (2) 处理年份进位，从 `day` 中逐次减去每年的天数，`year++`。
- (3) 处理月份进位，从 `day` 中逐次减去每月的天数，`month++`，必要时 `year++`。

根据这一思路，`CDate` 类的加法运算符重载函数，定义如下：

```
CDate CDate::operator+(int n) const{
    CDate dt = *this; dt.day = day + n;
    int mode = getDaysOfYear(dt.year); //处理年份进位
    while(dt.day > mode) {
        dt.day -= mode; dt.year++;
        mode = getDaysOfYear(dt.year);
    }
    mode = getDaysOfMonth(dt.year, dt.month); //处理月份进位
    while(dt.day > mode) {
```

```

        dt.day -= mode; dt.month++;
        if(dt.month>12) { dt.year++; dt.month = 1; }
        mode = getDaysOfMonth(dt.year, dt.month);
    }
    return dt;
}

```

2.2.8 扩展 CDate 类的+=、-=、++、--运算

在 CDate 类的加法和减法运算基础上, 实现-n、+=、-=、++、--等运算就很容易了, 举手之劳, 何乐而不为。

```

CDate CDate::operator-(int n) const {
    CDate dt = *this; dt.day = day - n;
    int mode = getDaysOfYear(dt.year-1);
    while(dt.day < 1) {
        dt.day += mode; dt.year--;
        mode = getDaysOfYear(dt.year-1);
    }
    return dt+0;
}

CDate& CDate::operator+=(int n) {
    *this = *this + n; return *this;
}

CDate& CDate::operator-=(int n) {
    *this = *this -n; return *this;
}

CDate& CDate::operator++() {
    *this = *this + 1; return *this;
}

CDate& CDate::operator++(int) {
    CDate dt = *this; *this = *this + 1; return dt;
}

CDate& CDate::operator--() {

```

```
*this = *this - 1; return *this;
}
CDate& CDate::operator--(int) {
    CDate dt = *this; *this = *this - 1; return dt;
}
```

至此，日期问题核心的减法和加法代码编写完成。富有批判和怀疑精神的程序员往往会自我反省：程序实现的对吗？有没有漏洞？这就要求我们掌握单元测试技术。

3. 单元测试保障代码质量

所谓单元测试（Unit Test），其实就是一段代码，功能是检验一个函数或者一个类的功能是否符合预期。程序员必须养成对代码进行单元测试的习惯。为了保障核心代码的质量，有必要对日期类 `CDate` 的加减法运算符重载函数进行单元测试。

在 `Visual Studio 2010` 中进行单元测试，需完成如下步骤的工作：（1）创建一个 `C++` 单元测试项目。（2）在单元测试项目中引入被测试代码。（3）编写单元测试代码。（4）执行单元测试。

3.1 创建一个 C++ 单元测试项目

右键单击“解决方案”，在弹出的菜单中选择【添加/新建项目】。点击 `Visual C++` 分类，选择【测试】子类下的【测试项目】，给项目取名为 `CDateTest`，然后点击确定，参见图 27 中左侧的“添加新项目”窗口。

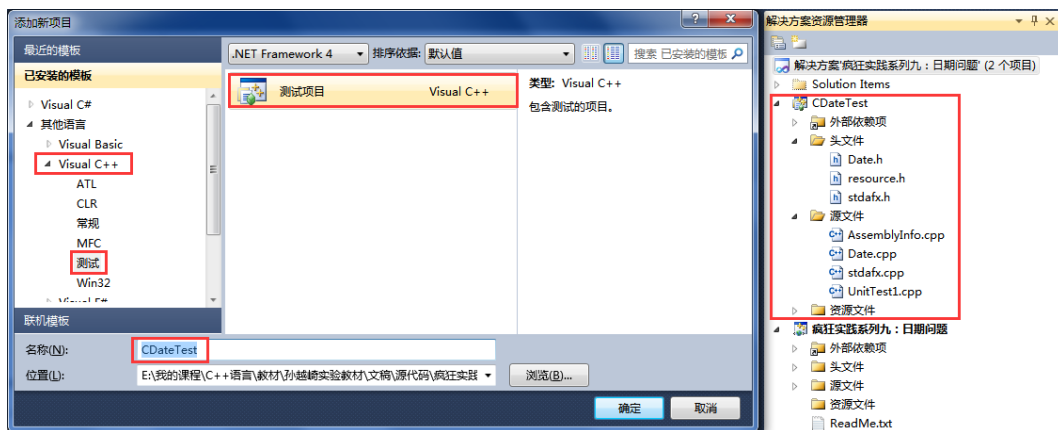


图 27 添加单元测试项目

此时，Visual Studio 会创建一个包含单元测试的项目 CDateTest，参见图 27 中右侧的“解决方案资源管理器”窗口。

右键单击 CDateTest 项目，选择【属性】，将【配置属性】下的【公共语言运行时支持】设为“公共语言运行时支持(/clr)”，参见图 28。否则，在测试代码中#include 被测试项目的 C++头文件时，就会产生编号为 C4956 和 C4959 的编译错误。

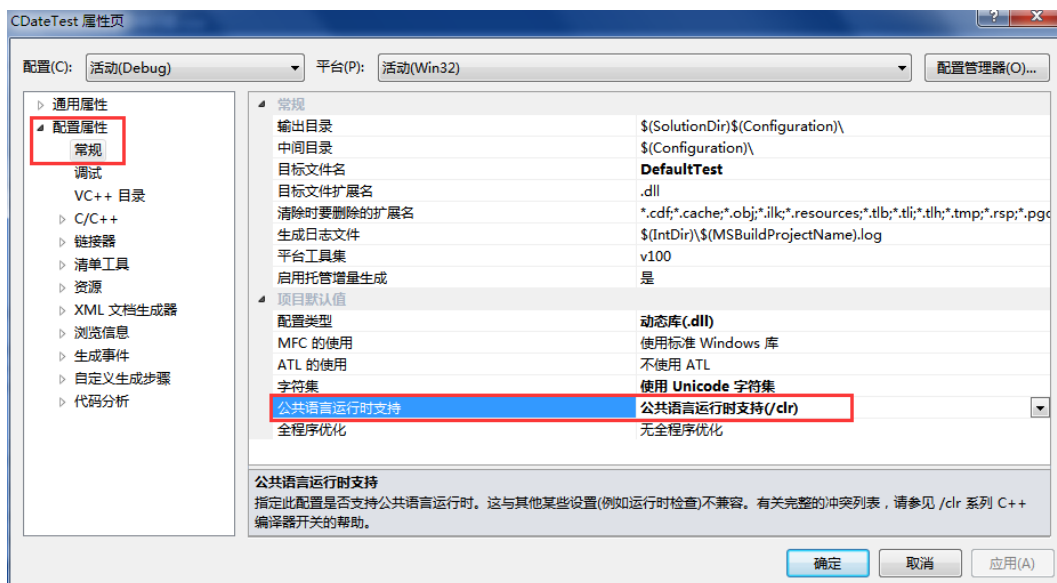


图 28 在测试项目属性中设定“公共语言运行时支持(/clr)”

3.2 在单元测试项目中引入被测试代码

在解决方案资源管理器中，右键单击单元测试的项目 CDateTest，在弹出菜单中选择【添加/现有项】，在弹出的文件浏览窗口中找到 CDate 类的头文件 Date.h 和源文件 Date.cpp，选择并添加之。

3.3 编写单元测试代码

编写单元测试代码。打开 CDateTest 项目下的源文件：UnitTest1.cpp，在图 29 所示的代码窗口中添加代码。对于测试 CDate 类的减法和加法运算而言，

(1) 包含 CDate 类的头文件“date.h”，如图 29 所示。

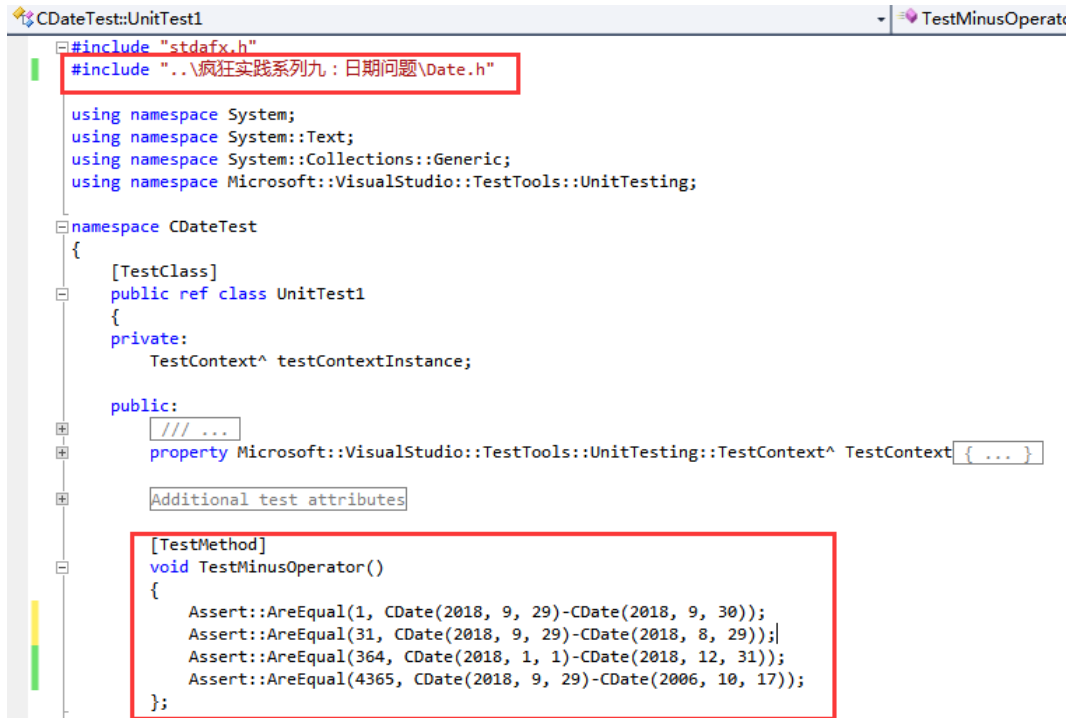


图 29 单元测试的源文件 UnitTest1.cpp

(2) 添加测试函数 TestMinusOperator 和 TestAddOperator，代码如下：

```

[TestMethod]
void TestMinusOperator() {
    Assert::AreEqual(1, CDate(2018,9,29)-CDate(2018,9,30));
    Assert::AreEqual(31, CDate(2018,9,29)-CDate(2018,8,29));
    Assert::AreEqual(364, CDate(2018,1,1)-CDate(2018,12,31));
    Assert::AreEqual(4365, CDate(2018,9,29)-CDate(2006,10,17));
}

[TestMethod]
void TestAddOperator(){
    Assert::IsTrue(CDate(2019,1,1)==(CDate(2018,12,31)+1));
    Assert::IsTrue(CDate(2018,9,29)==(CDate(2006,10,17)+4365));
}
    
```

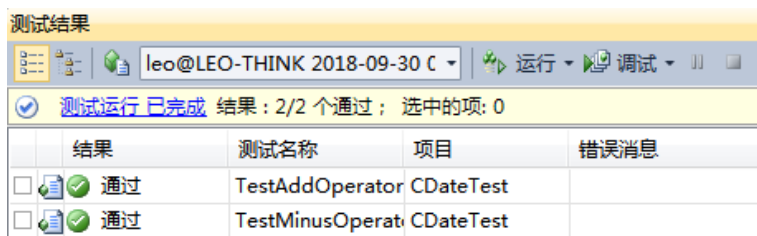
代码中，**Assert::AreEqual** 是判定两个表达式是否相等的断言，第一个表达式是期望值，第二个是被测试的表达式，如果测试值等于期望值，则断言成立测试通过，否则测

试不通过。**Assert::IsTrue** 是判断命题表达式是否成立，如果成立则测试通过，否则测试不通过。

Assert 类提供了一系列断言方法，例如 **AreEqual**、**AreNotEqual**、**AreSame**、**IsTrue**、**IsFalse**、**IsNull**、**IsNotNull**，用于判断相等、不等、相同、为真、为假、为空、不空等断言是否成立，具体使用方法请读者自行尝试学习并应用于单元测试中。

3.4 执行单元测试

执行单元测试。点击【测试】菜单，选择【运行/解决方案中的所有测试】。得到的测试结果如图 30 所示，说明测试通过。还可以根据预期假设添加更多的断言，或者添加其他的测试函数，注意测试函数要有 **[TestMethod]** 标记。





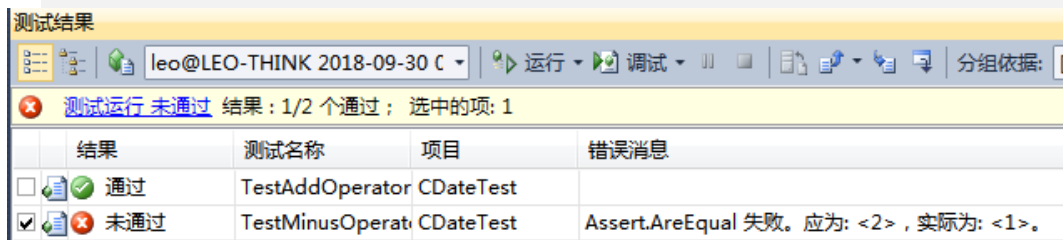
结果	测试名称	项目	错误消息
<input type="checkbox"/>  通过	TestAddOperator	CDateTest	
<input type="checkbox"/>  通过	TestMinusOperator	CDateTest	

图 30 CDateTest 单元测试通过

如果把代码稍作修改，使得某个断言不成立，则测试会失败，如图 31 所示。

```
Assert::AreEqual(2, CDate(2018, 9, 29)-CDate(2018, 9, 30));
```





结果	测试名称	项目	错误消息
<input type="checkbox"/>  通过	TestAddOperator	CDateTest	
<input checked="" type="checkbox"/>  未通过	TestMinusOperator	CDateTest	Assert.AreEqual 失败。应为: <2>, 实际为: <1>。

图 31 CDateTest 单元测试不通过

一般情况下，要求在代码每次修改后运行所有单元测试，必须保证所有单元测试都能够通过，称为回归测试。单元测试和回归测试，是保障代码质量的重要手段。

4. 完整代码组织结构

```
//源文件: main.cpp
#include "stdafx.h"
#include "Date.h"
#include <iostream>
using namespace std;
int main() { ... }

//头文件: Date.h
#pragma once
#include <iostream>
using namespace std;
const int DAYS[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
const int DaysOfLeapYear = 366;
const int DaysOfNonLeapYear = 365;
class CDate{
public:
    CDate(int y=1, int m=1, int d=1):year(y), month(m), day(d){}
    int operator-(const CDate&) const;
    CDate operator+(int n) const;
    int operator>(const CDate&) const;
    bool operator==(const CDate&) const;
    CDate operator-(int n) const; //重载操作符-
    CDate& operator+=(int n); //重载操作符+=
    CDate& operator-=(int n); //重载操作符-=
    CDate& operator++(); //前置++
    CDate& operator++(int); //后置++
    CDate& operator--(); //前置--
    CDate& operator--(int); //后置--
    friend istream& operator>>(istream&, CDate&);
    friend ostream& operator<<(ostream&, const CDate&);
    int getDaysOfDate() const;
    static int getDaysOfMonth(int, int);
    static int getDaysOfYear(int);
    static bool isLeapYear(int);
private:
    void inputYear();
    void inputMonth();
```

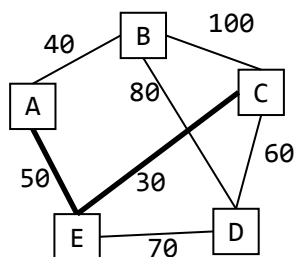
```
    void inputDay();  
private:  
    int year, month, day;  
};
```

```
//源文件: Date.cpp
```

```
#include "stdafx.h"  
#include "Date.h"  
int CDate::operator-(const CDate& date) const {...}  
CDate CDate::operator+(int n) const {...}  
int CDate::operator>(const CDate& date) const {...}  
bool CDate::operator==(const CDate& date) const {...}  
istream& operator>>(istream& ism, CDate& date) {...}  
ostream& operator<<(ostream& osm, const CDate& date) {...}  
.....
```

疯狂实践系列十：最短路径问题

题目：设有 N 个城市，如 $N=5$ 个城市分别是 ABCDE。给定如图 32 所示的城市路径地图，图中顶点表示各个城市，顶点之间的边表示两个城市之间连通的路径长度，请使用面向对象问题求解思维编写程序计算给定两个城市之间的最短路径。



城市"A"到城市"C":

- 最短路径: A->E->C
- 路径长度: 80

图 32 城市之间的最短路径示意图

1. 训练目标

- (1) 理解自顶向下、分而治之的问题求解思维。
- (2) 根据问题求解需求反演对象的属性和行为。
- (3) 理解使用二维数组表示城市路径图的方法。
- (4) 理解朴素的暴力穷举思路及其编程实现方法。
- (5) 理解 Dijkstra 算法思路及其编程实现方法。

2. 自顶向下分治

采用最直接的暴力穷举搜索法，找到两个城市之间的所有路径，然后计算每条路径的长度，从中找出最短路径。当然，这种问题求解思路比较笨拙，效率不高，但是对于语言学习者而言，这种笨拙的解题思路反倒是考较语言功底的绝佳机会。

2.1 第一层问题：main 函数

直观地，最短路径问题可以分解为三个任务：(1) 准备城市路径地图；(2) 计算两个城市间的最短路径；(3) 输出两个城市间的最短路径及其距离。

假设存在一种 **Map 类型** 的对象可以用来表达一组城市之间的路径地图，能够计算两个城市之间的最短路径，并且提供打印输出最短路径的行为能力。此外，还假设存在一种 **Path 类型** 的对象可以用于表达两个城市之间的一条路径。

如果假设成立，则编写 `main` 函数就非常简单。尝试编写代码如下：

```
int main() {
    string citys[] = {"A", "B", "C", "D", "E"}; //城市名
    int nCitys = sizeof(citys)/sizeof(string); //城市个数
    Map cityMap(nCitys, citys);
    buildMap(cityMap);
    cityMap.print();
    Path path = cityMap.getShortestPath("A", "E");
    cout<<"城市 A->E 的最短路径为: "
    cityMap.printPath(path);
    cout<<"路径长度为: "<<cityMap.getPathDist(path)<<endl;
    return 0;
}
```

根据 `main` 函数实现的需求，可以反向推理得出表示城市路径地图的 **Map** 类至少具有如下的公有成员函数：

(1) **Map** 构造函数，用于定义城市路径地图对象产生的行为。原型为，

```
Map(int, string);
```

(2) `getShortestPath` 成员函数，查询给定两个城市的最短路径。原型为，

```
Path getShortestPath(string, string) const;
```

(3) `printPath` 成员函数，用于输出一条路径。原型为，

```
void printPath(const Path&) const;
```

(4) `getPathDist` 成员函数，用于计算给定路径的长度距离。原型为，

```
double getPathDist(const Path& path) const;
```

(5) `print` 成员函数，用于输出城市地图的直接距离矩阵。原型为，

```
void print() const;
```

此外，还需要定义一个 `buildMap` 函数（注意，不是 **Map** 的成员函数），用于根据图 32 设定 **Map** 对象中城市之间的直接连通距离。函数原型为，

```
void buildMap(Map& cityMap);
```

现在最短路径问题就归结为定义 Map 类和 buildMap 函数。

2.2 第二层问题：buildMap 函数

编程第一准则：从简单的任务入手。buildMap 函数相对比较简单，其功能是根据图 32 所示的城市路径示意图设定 Map 对象中城市之间的直接连通距离。

为了解决问题，这里假设 Map 对象有一个成员函数叫做 setDist，可以设定两个城市之间的直接连通距离，成员函数原型为：

```
void setDist(string city1, string city2, double dist);
```

其中，city1 和 city2 分别是第一个和第二个城市的名字，dist 是直接连通距离。如果不连通则默认距离为 0。

利用 Map 对象的 setDist 成员函数，实现 buildMap 函数很方便。代码如下：

```
void buildMap(Map& cityMap){  
    cityMap.setDist("A", "B", 40);  
    cityMap.setDist("A", "E", 50);  
    cityMap.setDist("B", "C", 100);  
    cityMap.setDist("B", "D", 80);  
    cityMap.setDist("C", "D", 60);  
    cityMap.setDist("C", "E", 30);  
    cityMap.setDist("D", "E", 70);  
}
```

2.3 第二层问题：定义 Map 类

类 = 属性特征 + 行为特征 = 数据成员 + 成员函数 = 公有成员 + 私有成员。因此，定义类就需要确定类的属性和行为能力，从而确定类的数据成员和成员函数，进而确定一个类对外提供的公有成员和隐藏起来的私有成员。

2.3.1 Map 类的需求分析

Map 类需要完成的任务包括：产生 Map 对象；计算两个城市间的最短路径；输出两个城市间的最短路径及其距离。要完成这些任务，Map 对象必须知道城市及其这些城市之间的直接连通距离（图 32），故可以推断 Map 类应具有如下属性：

- （1）保存城市个数的属性：假设数据成员叫做 **nCitys**。
- （2）保存城市名字的属性：假设数据成员叫做 **citys**。
- （3）保存城市之间直接连通距离的属性：假设数据成员叫做 **dists**。

2.3.2 保存城市之间的直接连通距离

属性 **dists**，即 Map 类的数据成员 **dists**，用于保存城市之间直接连通距离。

对于图 32 中 5 个城市的直接连通距离，可表示成表格形式。具体做法是：

- （1）首先对城市从 0 开始按递增顺序编号。“A”编号 0，“B”编号 1，……。

城市	A	B	C	D	E
编号	0	1	2	3	4

- （2）然后，建立 5×5 的表格，把城市之间的直接连通距离按编号填充到表格当中。

填充方法：假设城市 C1 到 C2 的距离为 d ，则以 C1 的编号为行，以 C2 的编号为列，定位到一个表格单元，将距离 d 填入该表格单元；同时，以 C2 的编号为行，以 C1 的编号为列，定位到一个表格单元，将距离 d 填入该表格单元。

例如，城市“A”到“E”距离 50，而“A”的编号为 0，“E”的编号为 4，则在 5×5 表格的第 0 行第 4 列填入 50，第 4 行第 0 列也填入 50，表示距离是对称的。

依此类推，可以形成图 33 所示的二维表格。表格中每个元素表示了行列编号对应的两个城市的直接连通距离，0 表示两个城市不连通。例如，第 3 行第 1 列的 80，表示编号为 3 的城市与编号为 1 的城市之间，直接连通距离为 80。

编号	0	1	2	3	4
0	0	40	0	0	50
1	40	0	100	80	0
2	0	100	0	60	30
3	0	80	60	0	70
4	50	0	30	70	0

图 33 通过表格表示城市之间的直接连通距离

不言而喻，可用二维数组来保存城市之间的直接连通距离。考虑到城市个数不确定，需要动态申请二维数组，故通过指针保存二维动态数组的地址：

```
double** dists;
```

2.3.3 定义 Map 类的初略框架

整理目前对 Map 类的属性和行为能力的假设，尝试定义 Map 类如下：

```
class Path { }; //假设存在辅助类 Path，表达城市路径
class Map {
public:
    Map(int, string[]);
    ~Map();
    void setDist(string, string, double);
    Path getShortestPath(string, string) const;
    void print() const;
    void printPath(const Path&) const;
    double getPathDist(const Path&) const;
private:
    int nCitys;
    string *citys;
    double** dists;
};
```

2.3.4 定义 Map 类的构造和析构函数

Map 类的构造函数，用于定义 Map 对象产生时的行为。根据上文分析，Map 对象在产生时应该完成如下的初始化工作：

- (1) 初始化 nCitys，设定城市个数。
- (2) 初始化 citys，为其申请保存城市名字的动态数组并设定城市名称。
- (3) 初始化 dists，为其申请二维动态数组保存城市之间的直接连通距离，初始值全部设置为 0，表示初始时城市之间不连通。

根据上述要求，尝试编写 Map 类的构造函数，代码如下：

```
Map::Map(int n, string citynames[]) : nCitys(n) {
```

```

citys = new string[nCitys];
dists = new double*[nCitys];
for(int i=0; i<nCitys; i++) {
    citys[i] = citynames[i];
    dists[i] = new double[nCitys];
    for(int j=0; j<nCitys; j++) dists[i][j]=0;
}
}

```

构造函数中申请了动态数组，需要在析构函数中释放。析构函数代码如下：

```

Map::~~Map() {
    delete[] citys;
    for(int i=0; i<nCitys; i++) delete[] dists[i];
    delete[] dists;
}

```

2.3.5 定义 setDist 和 print 成员函数

setDist 成员函数用于设定两个城市之间的直接连通距离，比较简单，先行编写。

城市的直接连通距离保存在数据成员 dists 中，是一个二维动态数组，访问二维数组元素需要指定两个下标，故首先需要根据城市的名字计算其编号。

假设 Map 类有成员函数 getCityIndex，用于根据城市名字计算城市编号。由于该成员函数仅服务于 Map 类，故将其声明为私有成员函数。代码如下：

```

int Map::getCityIndex(string city) const {
    for(int i=0; i<nCitys; i++)
        if(citys[i] == city) return i;
    return -1;
}

```

有了假设的 getCityIndex 函数，定义 setDist 非常轻松，代码如下：

```

void Map::setDist(string city1, string city2, double dist) {
    int i= getCityIndex(city1);
    int j = getCityIndex(city2);
    if(i!=j) dists[i][j] = dists[j][i] = dist;
}

```

```
}
```

此外，`print` 成员函数，用于输出城市地图的直接连通距离。代码如下：

```
void Map::print() const {
    cout<<"输出地图: "<<endl;cout<<"\t";
    for(int i=0; i<nCitys; i++) { cout<<citys[i]<<"\t"; }
    cout<<endl;
    for(int i=0; i<nCitys; i++) {
        cout<<citys[i]<<"\t";
        for(int j=0; j<nCitys; j++) { cout<<dists[i][j]<<"\t"; }
        cout<<endl;
    }
}
```

至此，一路下来我们的编程思维和过程比较顺畅，求解问题有这样的需求，所有我们假设存在那样的函数或者类来满足需求，然后给出函数或者类的定义。

接下来就需要面对最短路径计算的核心问题，定义 `Map` 类的 `getShortestPath` 成员函数，骨头有点硬，比较难啃。

2.4 暴力穷举计算最短路径：getShortestPath 函数

计算两个城市之间的最短路径，采用暴力穷举法：给定城市 `city1` 和 `city2`，以 `city1` 为起点遍历城市地图，寻找所有从 `city1` 到 `city2` 的路径。

每当找到一条从 `city1` 到 `city2` 的路径，就立即计算该路径的长度距离，如果该路径比当前最短路径更短，则将其保存为当前最短路径。

最后，当暴力穷举完 `city1` 到 `city2` 的所有路径时，当前最短路径就是我们最终要求的从 `city1` 到 `city2` 的最短路径。

2.4.1 暴力穷举所有路径

对于图 34(a)所示的城市路径示意图，假设要找出以"B"起点且以"E"终点的所有路径，则暴力穷举从起点"B"开始，然后执行如下的步骤（图 34(b)）：

- （1）寻找与"B"直接连通的第 1 个城市，得到"A"；
 - （1.1）寻找与"A"直接连通的第 1 个城市，得到"E"；

遇到终点"E", 说明找到了一条路径 **B->A->E**。不再沿着"E"继续往下找, 回退到"A";

(1.2) 寻找与"A"直接连通的第 2 个城市, 找不到;

不再沿着"A"继续往下找, 回退到"B";

(2) 寻找与"B"直接连通的第 2 个城市, 得到"C";

(2.1) 寻找与"C"直接连通的第 1 个城市, 得到"D";

(2.1.1) 寻找与"D"直接连通的第 1 个城市, 得到"E";

遇到终点"E", 说明找到了一条路径 **B->C->D->E**。不再沿着"E"继续往下找, 回退到"D";

(2.1.2) 寻找与"D"直接连通的第 2 个城市, 找不到;

不再沿着"D"继续往下找, 回退到"C";

(2.2) 寻找与"C"直接连通的第 2 个城市, 得到"E";

遇到终点"E", 说明找到了一条路径 **B->C->E**。不再沿着"E"继续往下找, 回退到"C";

(2.3) 寻找与"C"直接连通的第 3 个城市, 找不到;

不再沿着"C"继续往下找, 回退到"B";

(3) 寻找与"B"直接连通的第 3 个城市, 得到"D";

(3.1) 寻找与"D"直接连通的第 1 个城市, 得到"C";

(3.1.1) 寻找与"C"直接连通的第 1 个城市, 得到"E";

遇到终点"E", 说明找到了一条路径 **B->D->C->E**。不再沿着"E"继续往下找, 回退到"C";

(3.1.2) 寻找与"C"直接连通的第 2 个城市, 找不到;

不再沿着"C"继续往下找, 回退到"D";

(3.2) 寻找与"D"直接连通的第 2 个城市, 得到"E";

遇到终点"E", 说明找到了一条路径 **B->D->E**。不再沿着"E"继续往下找, 回退到"D";

(3.3) 寻找与"D"直接连通的第 3 个城市, 找不到;

不再沿着"D"继续往下找, 回退到"B";

(4) 寻找与"B"直接连通的第 4 个城市, 找不到, 暴力穷举结束。

通过上述暴力穷举过程，就找到了以城市"B"为起点且以城市"E"为终点的所有路径，对应于图 34(b)中以粗实线表示的五条路径：B->A->E、B->C->D->E、B->C->E、B->D->C->E、B->D->E。

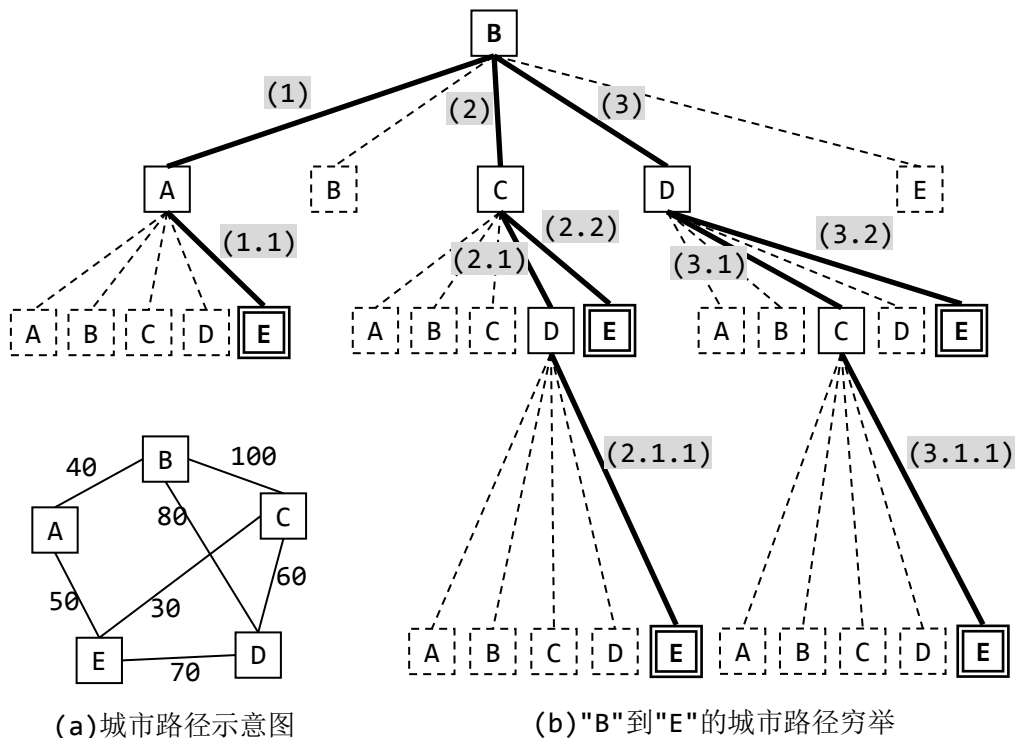


图 34 以城市"B"为起点"E"为终点的所有城市路径

上述暴力穷举所有路径的过程，构成了图 34(b)所示的一棵以"B"为根节点的分层搜索树，每条城市路径都对应于从根节点"B"开始向下的一个分支选择路径。如，城市路径 B->C->D->E，就对应于(2)->(2.1)->(2.1.1)的一个分支选择路径。

在搜索树中，除叶子节点之外的每个树节点，都扩展伸出了 5 个分支，对应于 5 个子节点，其中虚线分支表示扩展失败；实线分支表示扩展成功。

假设节点 X 是根节点或者某个中间节点，节点 Y 是从节点 X 扩展伸出的 5 个分支节点之一，则从节点 X 成功扩展出节点 Y 需要满足两个条件：

- (1) 节点 Y 与节点 X 之间直接连通；
- (2) 节点 Y 在节点 X 的当前路径上不曾出现过。

搜索过程从根节点"B"开始沿着树的深度方向展开，首先找到与"B"直接连通的第 1 个节点"A"，然后以"A"为当前节点，找到与"A"直接连通的第 1 个节点"E"。

如果当前节点是终止节点"E"，则表明沿着该路径已经走到尽头，找到一条路径；此时，停止扩展节点"E"，回退到该节点的父节点继续搜索，这里就回退到节点"A"。

然后继续沿着"A"，搜索与"A"直接连通的第 2 个节点。在城市地图中，与"A"直接连通的城市有两个："B"和"E"，其中"E"已经找过了，"B"在当前路径上已经出现过，此外已经找不到与"A"直接连通的城市了，故停止扩展节点"A"，回退到节点"A"的父节点"B"，然后继续沿着节点"B"，搜索与"B"直接连通的第 2 个节点……。

这种沿着搜索树的深度方向不断向下搜索，直到碰壁，然后才回退到上层父节点继续搜索的过程，专业术语称为“回溯 (BackTracking)”。

2.4.2 回溯搜索的递归编程框架

搜索树的回溯，本质上是一个递归过程：父节点的回溯，等价于所有子节点的回溯。假设函数 `backTracking(city1, city2)` 能够回溯搜索从 `city1` 开始到 `city2` 终止的所有路径，则使用 `backTracking` 函数能够回溯搜索从 `city1` 的任意子节点开始到 `city2` 终止的所有路径。而这两者存在如下的等价关系：

```
backTracking("B","E") = backTracking("A","E")
                        + backTracking("C","E")
                        + backTracking("D","E")
```

因此，使用递归实现回溯搜索函数 `backTracking` 比较容易，代码框架如下：

```
void backTracking(int city1, int city2) {
    if(city1 == city2) { /*找到一条路径，更新最短路径，终止回溯*/ }
    for(int city=0; city<nCitys; city++) { //回溯子节点
        if((city 不等于 city1) && (city 和 city1 直接连通)
            && (city 在当前路径中没有出现过))
            backTracking(city, city2);
    }
}
```

考虑到 `backTracking` 函数需要访问城市之间的直接连通关系 (`Map` 对象的数据成员 `dists`)，故将 `backTracking` 函数设计为 `Map` 类的私有成员函数是合适的。

此外，在回溯搜索 `backTracking` 成员函数中，还有几项任务必须完成：

(1) 需要比较当前路径和当前最短路径的长度距离，如果当前路径比当前最短路径更短，则将当前最短路径更新为当前路径；

(2) 需要判定城市 **city** 是否在当前路径中出现过。

显然，在 **backTracking** 成员函数中必须记住：当前路径、当前最短路径及其长度。实际上更基础的一个问题是：路径如何表示，即定义 **Path** 类。

2.4.3 定义 Path 类描述城市路径

假设有 **N** 个城市，城市编号从 **0~N-1**。为了简化问题，把任意城市路径表示为长度为 **N** 的整数序列，如果在该城市路径中有 **k** 个城市，且按照城市路径顺序的城市编号序列为 **c₁->c₂->...->c_k->-1->...**，序列前 **k** 个元素为 **k** 个城市编号，剩余 **N-k** 个序列元素取值均为 **-1**。以图 34(a) 的城市地图为例 (**N=5**)，路径 **B->D->C->E** 可以表示为：

1	3	2	4	-1
---	---	---	---	----

根据这种城市路径的表示方法，需要定义 **Path** 类来表示城市路径。一条城市路径表示为长度为 **N** 的整数序列。不难推断，**Path** 类必须有两个属性：

(1) **nCitys**：表示城市路径序列的总长度。

(2) **path**：表示城市路径对应的序列数组，长度为 **nCitys**。

根据在 **Map** 类中的使用要求，**Path** 类的对象需要支持拷贝、赋值、序列中城市编号的访问等行为能力，尝试实现 **Path** 类的框架代码：

```
class Path {
public:
    Path(int n) : nCitys(n) {
        path = new int[nCitys];
        memset(path, -1, nCitys*sizeof(int));
    }
    Path(const Path& p) : nCitys(p.nCitys) {
        path = new int[nCitys];
        memcpy(path, p.path, nCitys*sizeof(int));
    }
    Path& operator=(const Path& p) {
        assert(nCitys == p.nCitys); //确保赋值相容
        if (this != &p) { memcpy(path, p.path, nCitys*sizeof(int)); }
```

```

        return *this;
    }
    ~Path() { delete[] path; }
    int& operator[](int i) const {
        assert(i>=0 && i<nCitys);
        return path[i];
    }
private:
    int nCitys;
    int *path; //城市编号序列
};

```

2.4.4 如何记住当前路径

当前路径，是从当前节点到其父节点，再到父节点的父节点，……，最终到根节点的一条路径。当前路径会随着回溯搜索过程发生变化，例如，

- (1) 初始情况下，当前节点是"B"，当前路径为{B};
- (2) 扩展搜索一次，则当前节点变成"A"，当前路径变成{B->A};
- (3) 再扩展搜索一次，则当前节点变成"E"，当前路径变成{B->A->E};
- (4) 再扩展搜索一次，由于当前节点是终止节点"E"，故停止扩展回退当前节点"E"的父节点"A"，即当前节点变成"A"，当前路径变成{B->A};
- (5) 再扩展搜索一次，发现"A"的所有子节点都搜索完成，找不到新的扩展节点，故停止扩展回退到当前节点"A"的父节点"B"，当前路径变成{B}
- (6) 再扩展搜索一次，则当前节点变成"C"，当前路径变成{B->C};
- (7) ……

观察从(1)~(7)的回溯搜索过程，每当向下扩展搜索时，当前节点发生变化，则当前路径也发生变化，在当前路径尾部添加当前节点；每当需要回退时，当前节点更新为其父节点，从当前路径尾部弹出最后一个节点。

根据这一需求，我们假设 **Path** 类还具有如下行为能力的成员函数：

- (1) **pushBack**: 向序列尾部添加城市编号;
- (2) **popBack**: 从序列尾部弹出城市编号;

(3) **isVisited**: 判定给定城市编号是否在序列中出现。

请读者自行在 **Path** 类头文件中添加这三个成员函数的声明，函数定义如下：

```
void Path::pushBack(int inx){
    assert(path[nCitys-1] == -1); //栈满
    for(int i=0; i<nCitys; i++){
        if(path[i]==-1) { path[i]=inx; break; }
    }
}

void Path::popBack(){
    assert(path[0]>=0); //栈空
    int i = nCitys-1;
    while(path[i]==-1) i--;
    path[i] = -1;
}

bool Path::isVisited(int cityNO) {
    for(int i=0; i<nCitys && path[i]!=-1; i++){
        if(path[i] == cityNO) return true;
    }
    return false;
}
```

在回溯搜索过程中，假设存在一个 **Path** 对象叫做 **currPath**，在恰当的时机调用其成员函数 **pushBack** 和 **popBack**，就可以记住当前路径：

(1) 每次扩展搜索发生时，导致当前节点发生变化，此时需要调用 **pushBack** 函数在当前路径的序列尾部加入当前节点。

(2) 如果当前节点是终止节点时，则需要调用 **popBack** 从当前路径的序列尾部弹出当前节点，回退到父节点；

(3) 如果当前节点的所有子节点扩展搜索都完成了，则需要调用 **popBack** 从当前路径的序列尾部弹出当前节点，回退到其父亲节点。

2.4.5 定义 backTracking 成员函数

至此，回溯搜索的细节基本上都已经澄清。对 **backTracking** 函数有如下假设：

(1) 假设参数 **currPath**，用于记住当前路径，需要在恰当的时机调用其成员函数 **pushBack** 和 **popBack**。

(2) 假设参数 `currShtPath`，用于记住当前最短路径，在当前路径比当前最短路径更短时，将当前最短路径更新为当前路径。

(3) 假设参数 `minDist`，用于记住当前最短路径的长度距离，在将当前最短路径更新为当前路径时，同时将 `minDist` 更新为当前路径的长度距离。

注意，当前路径、当前最短路径、当前最短路径的长度距离，需要在整个回溯搜索过程中共享唯一的一个副本，因此，将这三个参数定义为引用类型。

```
void Map::backTracking(int city1, int city2,
    Path&currPath, Path&currShtPath,double& minDist) const {
    currPath.pushBack(city1); //当前节点变化，记住当前路径
    if(city1 == city2) { /*遇到终止节点，找到路径*/
        double dist = getPathDist(currPath);
        if(dist<minDist) { minDist=dist; currShtPath=currPath; }
        currPath.popBack(); //当前节点为终止节点，回退到父节点
        return;
    }
    for(int city=0; city<nCitys; city++) {
        if((city != city1) && (dists[city1][city] > 0) &&
            !currPath.isVisited(city)) {
            backTracking(city,city2,currPath,currShtPath,minDist);
        }
    }
    currPath.popBack(); //所有子节点扩展完成，回退到父节点
}
```

这里，假设 `Map` 类存在 `getPathDist` 成员函数，可以计算路径的长度距离。

```
double Map::getPathDist(const Path& path) const {
    double dist = 0.0;
    for(int i=0; i<nCitys-1; i++) {
        if(path[i]>=0&&path[i+1]>=0)
            dist += dists[path[i]][path[i+1]];
    }
    return dist;
}
```

```
}
```

2.4.6 定义 getShortestPath 成员函数

利用 backTracking 成员函数，实现 getShortestPath 成员函数非常容易。

```
Path Map::getShortestPath(string city1, string city2) const {
    int c1 = getCityIndex(city1);
    int c2 = getCityIndex(city2);
    Path currPath(nCitys), currShtPath(nCitys);
    double minDist = DBL_MAX;
    backTracking(c1, c2, currPath, currShtPath, minDist);
    return currShtPath;
}
```

最后，再廓清寰宇，把边边角角的成员函数实现一下。大功告成，编译链接执行。

```
void Map::printPath(const Path& path) const {
    string spath = "";
    for(int i=0; i<nCitys; i++)
        if(path[i]>=0)spath += citys[path[i]] + "->";
    spath = spath.substr(0, spath.length()-2);
    cout<<spath<<endl;
}
```

2.5 完整的代码组织结构

```
//源文件: main.cpp
```

```
#include "stdafx.h"
#include "Map.h"
void buildMap(Map&);
int main() { ... }
void buildMap(Map& cityMap){ ... }
```

```
//头文件: Map.h
```

```
#pragma once
#include <string>
#include <iostream>
#include <cassert>
using namespace std;
```

```
class Path {
public:
    Path(int);
    Path(const Path&);
    ~Path();
    Path& operator=(const Path&);
    int& operator[](int) const;
    void pushBack(int);
    void popBack();
    bool isVisited(int);
private:
    int nCitys;
    int *path;
};

class Map{
public:
    Map(int, string []);
    ~Map();
    void setDist(string, string, double);
    int getCityIndex(string) const;
    void print() const;
    Path getShortestPath(string, string) const;
    void printPath(const Path&) const;
    double getPathDist(const Path&) const;
private:
    void backTracking(int, int, Path&, Path&, double&) const;
private:
    int nCitys;
    string* citys;
    double** dists;
};
```

//源文件: Map.cpp

```
#include "stdafx.h"
```

```
#include "Map.h"
```

……（具体实现代码请参考正文内容组织完成）

3. 更高效的 Dijkstra 算法

通过回溯法暴力穷举城市之间的所有路径，可以计算得到最短路径，但是这种思路并不高效。1959 年，计算机科学家 Dijkstra 提出一种计算更为高效的最短路径算法，用于计算从一个顶点到其余各顶点的最短路径，称为 Dijkstra 算法。

3.1 Dijkstra 算法思路

已知一个地图 G 由顶点集合 V 及其边的集合 E 组成，假设起始顶点是 v_0 ，目标是找到从 v_0 起始到所有其他顶点的最短路径。Dijkstra 算法假设：

(1) **集合 S** ，保存已经找到了最短路径的顶点。算法初始时，集合 S 中仅包含起始顶点 v_0 ，因为只有 v_0 最短路径已知且路径长度为 0。

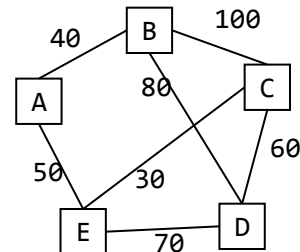
(2) **数组 $dist$** ，保存从起始顶点 v_0 到其余顶点的最短路径长度估计值。

算法每次从 $dist$ 数组中选择最短路径长度估计值最小的顶点，将其加入集合 S 中并重新估计 $dist$ 数组的最短路径值，重复该步骤直至所有顶点加入 S 。

以右边的城市地图为例，假设寻找从城市“B”出发到其余顶点的最短路径。步骤如下：

(1) 算法初始时集合 $S=\{B\}$ ，构造最短路径长度估计值 $dist$ 数组。

对于集合 S 中的顶点 B ，若存在能直到到达的边 $(B \rightarrow v)$ ，则把 $dist[v]$ 设置为 B 和 v 的直接连通距离，同时把不能直接到达的顶点路径长度估计为无穷大。



集合 S	$dist[A]$	$dist[B]$	$dist[C]$	$dist[D]$	$dist[E]$
$\{B\}$	40	0	100	80	∞

初始时情况不明，对最短路径长度的估计必然不会精确。但是，可以确信两点：

- 1) 最终的最短路径必然不会大于 $dist$ 数组里的估计值。
- 2) $B \rightarrow A$ 的最短路径长度是 40，且最短路径就是 $B \rightarrow A$ 。

(2) 更新集合 S 和 $dist$ 数组

此时由于 $B \rightarrow A$ 的最短路径已知， $dist[A]=40$ 确信不会变化，所以需要把顶点 A 从 $V-S$ 集合中移到已知最短路径的顶点集合 S 中， $S=\{B,A\}$ 。

顶点 A 的引入,使得我们对最短路径长度的估计值有了更多依据。在图 34(a)从顶点 A 引出边(A->E),以 A 为中介再次估计 B->E 的最短路径长度为:

$$\text{dist}[A] + (A,E) = 40 + 50 = 90 < \text{dist}[E] = \infty$$

因此,需要将 $\text{dist}[E]$ 更新为 90,顶点 A 到其他顶点没有直达路径,不会引起其他顶点的最短路径长度估计发生变化。此时的最短路径长度估计值 dist 为:

集合 S	$\text{dist}[A]$	$\text{dist}[B]$	$\text{dist}[C]$	$\text{dist}[D]$	$\text{dist}[E]$
{BA}	40	0	100	80	90

(3) 继续更新集合 S 和 dist 数组

从 dist 数组中找出最小值 $\text{dist}[D]=80$,可以确信: B->D 的最短路径长度为 80,且最短路径是 B->D。因为根据 dist 数组的估计值,不可能找到任何中介顶点 v,使得 B->v->D 的路径长度比 B->D 的路径长度更短,故 B->D 的估计准确。

将顶点 D 移到集合 S 中,继续更新 dist 数组。观察图 34(a),从顶点 D 引入三条边 (D,B)、(D,C)、(D,E),因为顶点 B 已经在集合 S 中,故不考虑边 (D,B)。

边 (D,C) 的引入会导致对 B->C 的最短路径估计更精确。已知 $\text{Dist}[D]=80$,即确信 B->D 的最短路径长度为 80,则以 D 为中介顶点, B->C 的最短路径估计为:

$$\text{dist}[D] + (D,C) = 80 + 60 = 140 > \text{dist}[C] = 100$$

边 (D,E) 的引入会导致对 B->E 的最短路径估计更准确。B->E 的最短路径估计为:

$$\text{dist}[D] + (D,E) = 80 + 70 = 150 > \text{dist}[E] = 90$$

根据上述分析,此时的最短路径长度估计值 dist 为:

集合 S	$\text{dist}[A]$	$\text{dist}[B]$	$\text{dist}[C]$	$\text{dist}[D]$	$\text{dist}[E]$
{BAD}	40	0	100	80	90

(4) 继续更新集合 S 和 dist 数组

从 dist 数组中找出最小值 $\text{dist}[E]=90$,将顶点 E 加入集合 S 中更新 dist 数组。图中顶点 E 引出三条边 (E,A)、(E,C)、(E,D),由于 A 和 D 均已加入集合 S 中表明其最短路径已知,故不考虑边 (E,A) 和 (E,D)。

引入边 (E,C) 可能会导致 B->C 的最短路径估计更准确。B->C 的最短路径估计为:

$$\text{dist}[E] + (E,C) = 90 + 30 = 120 > \text{dist}[C] = 100$$

故确信: B->C 的最短路径长度为 100,最短路径为 B->C。更新 dist 数组为:

集合 S	dist[A]	dist[B]	dist[C]	dist[D]	dist[E]
{BADE}	40	0	100	80	90

最后，将剩下的顶点 C 加入 S，最短路径长度为 $\text{dist}[C]=100$ 。

3.2 Dijkstra 算法 main 函数

Dijkstra 最短路径算法的 main 函数与暴力穷举算法的基本相同。同样假设存在 Map 对象，可以计算给定起始顶点到其余顶点的最短路径。代码如下：

```
//源文件：main.cpp
#include "stdafx.h"
#include "Map.h"
void buildMap(Map& Map);
int main() {
    string citys[] = {"A", "B", "C", "D", "E"}; //城市名
    int nCitys = sizeof(citys)/sizeof(string); //城市个数
    Map cityMap(nCitys, citys);
    buildMap(cityMap);
    cityMap.print();
    cityMap.getShortestPath("A");
    cout<<"最短路径为: "; cityMap.printPath("A", "C");
    cout<<"路径长度: "; cout<<cityMap.getPathDist("A", "C")<<endl;
    return 0;
}
void buildMap(Map& cityMap) { /*与前文同*/ }
```

3.3 定义 Map 类

与暴力穷举算法代码类似，假设 Map 类具有数据成员：nCitys、citys 和 dists；此外还假设了三个数据成员，用于记录 Dijkstra 算法中的过程和结果：

- (1) 数据成员 citySets：用于记录 Dijkstra 算法中的集合 S。
- (2) 数据成员 shtDists：用于记录 Dijkstra 算法中的 dist 数组。
- (3) 数据成员 prevCitys：用于记录每个顶点的最短路径的前一个顶点。

就物理意义而言，计算最短路径只是要获得计算结果，不应该修改 Map 类的数据成员，但是 citySets、shtDists、prevCitys 是用于记录计算过程和结果的数据成员，在最短路径计算过程中必定会被修改，故将这三个数据成员声明为 mutable。

此外，还假设 Map 类具有一系列成员函数：

- (1) setDist、getCityIndex、print 作用和代码与暴力穷举算法相同。
- (2) getShortestPath: 用于计算从 city1 开始到其余顶点的最短路径。
- (3) printPath: 用于打印输出 city1 到 city2 的最短路径。
- (4) getPathDist: 用于计算从 city1 到 city2 的最短路径长度。
- (5) initDijkstra: 用于初始化 citySets、shtDists、prevCitys。
- (6) findMinDist: 用于从 shtDists 数组中找到最短路径估计值最小的顶点。

//头文件: Map.h

```
#pragma once
#include "string"
#include <iostream>
#include <cassert>
using namespace std;
class Map{
public:
    Map(int n, string citynames[]);
    ~Map();
    void setDist(string city1, string city2, double dist);
    int getCityIndex(string city) const;
    void print() const;
    void getShortestPath(string city1) const;
    double getPathDist(string city1, string city2) const;
    void printPath(string city1, string city2) const;
private:
    void initDijkstra(int c1) const;
    int findMinDist() const;
private:
    mutable bool * citySets;
    mutable double *shtDists;
    mutable int* prevCitys;
private:
    int nCitys;
    string* citys;
    double** dists;
};
```



```
//源文件: Map.cpp
#include "stdafx.h"
#include "Map.h"
Map::Map(int n, string citynames[]) { /*根据前文自行组织*/ }
Map::~Map() { /*根据前文自行组织*/ }
void Map::setDist(string city1, string city2, double dist)
{ /*与前文同*/ }
int Map::getCityIndex(string city) const { /*与前文同*/ }
void Map::print() const { /*与前文同*/ }
double Map::getPathDist(string city1, string city2) const {
    int c2 = getCityIndex(city2);
    return shtDists[c2];
}
……//其他成员函数定义参见下文。
```

3.3.1 定义 getShortestPath 成员函数

成员函数 `getShortestPath`，用于计算从 `city1` 开始到其余顶点的最短路径。

```
void Map::getShortestPath(string city1) const {
    int c1 = getCityIndex(city1);
    //1. 初始化数据成员 citySets、shtDists、prevCitys
    initDijkstra(c1);
    int u = c1; //u 记录最短路径估计值最小的顶点，初始为 c1
    double minDist = 0.0; //当前最短路径最小值，初始为 0.0
    for(int i=0; i<nCitys; i++) {
        //2. 更新记录最短路径估计值的 shtDists 数组
        updateShtDists(u, minDist);
        //3. 从 shtDists 数组中找出最短路径最小的顶点
        u = findMinDist(); minDist = shtDists[u];
    }
}
```

在 `getShortestPath` 成员函数中，主要工作包括：初始化三个数据成员、反复更新 `shtDists` 数组、从 `shtDists` 数组中找出最短路径最小的顶点，假设存在三个成员函数用于完成这三份工作：`initDijkstra`、`updateShtDists`、`findMinDist`。

```

void Map::initDijkstra(int c1) const {
    for(int i=0; i<nCitys; i++) {
        citySets[i]=false;  shtDists[i]=DBL_MAX;  prevCitys[i]=-1;
    }
    shtDists[c1] = 0.0;
}

void Map::updateShtDists(int u, double minDist) const {
    citySets[u] = true;
    for(int k=0; k<nCitys; k++) {
        if(!citySets[k] && dists[u][k]>0
            && minDist+dists[u][k]<shtDists[k]) {
            shtDists[k] = minDist + dists[u][k];
            prevCitys[k] = u;
        }
    }
}

int Map::findMinDist() const {
    int u = -1;  double minDist = DBL_MAX;
    for(int k=0; k<nCitys; k++) {
        if(!citySets[k] && shtDists[k]<minDist) {
            minDist = shtDists[k];  u = k;
        }
    }
    return u;
}

```

3.3.2 定义 printPath 成员函数

最短路径计算完成后，需要 `printPath` 成员函数输出两个城市之间的最短路径。

Dijkstra 算法通过 `prevCitys` 数组记录每个城市的最短路径上的前一个城市编号。

例如，下面的 `prevCitys` 数组就记录了每个城市的前一个城市的编号。

A	B	C	D	E
-1	0	4	1	0

- (1) -1 表示是起始顶点。A 的前一个城市编号为-1，说明 A 是起始顶点。
- (2) B 的前一个城市编号为 0，表示最短路径上 B 的前一个城市为 A。
- (3) C 的前一个城市编号为 4，表示最短路径上 C 的前一个城市为 E。
- (4) D 的前一个城市编号为 1，表示最短路径上 D 的前一个城市为 B。
- (5) E 的前一个城市编号为 0，表示最短路径上 E 的前一个城市为 A。

假设要输出 A->C 的最短路径，则从 C 开始反复找前一个城市编号，直到-1 即到达起始顶点。由于 C 的城市编号为 2，则查找 A->C 的最短路径过程如下：

```
2 <- prevCity[2]=4 <- prevCity[4]=0 <- prevCitys[0]=-1
```

根据上述分析，实现 printPath 成员函数，代码如下：

```
void Map::printPath(string city1, string city2) const {
    int c2 = getCityIndex(city2);
    string path = citys[c2];
    int city = c2;
    while(prevCitys[city]>=0) {
        city = prevCitys[city];
        path = citys[city] + "->" + path;
    }
    cout<<path<<endl;
}
```

至此，最短路径计算的 Dijkstra 算法实现大功告成，编译链接执行。细心的读者不妨比较一下暴力穷举搜索和 Dijkstra 算法的执行效率。

疯狂实践系列十一：方程求根问题

题目：在科学研究和工程实践中经常需要求解非线性方程 $f(x)=0$ 。给定任意方程，使用等间隔搜索法、二分搜索法、牛顿迭代法、弦截法四种方法求该方程在区间 $[a,b]$ 上的近似实根，相互印证求根结果的正确性、精度和适用场合。例如，

$$f(x) = x^3 - 3.2x^2 + 1.9x + 0.8 = 0.0$$

求该非线性方程在区间 $[-5, 5]$ 上的所有实根。

1. 训练目标

- (1) 理解自顶向下、分而治之的问题求解思维。根据问题求解需求分析推理对象的属性和行为，迭代修正类的设计。
- (2) 掌握函数指针及其使用方法和 `typedef` 语法。
- (3) 理解类的继承和多态，掌握虚函数及纯虚函数的使用方法。理解 `Solver` 类的 `solve` 函数与 `soleSolve` 函数的关系。
- (4) 学习 `vector` 容器对象的建立、插入和访问方法。
- (5) 学习非线性方程的四种具体求根方法和代码实现。

2. 自顶向下分治

2.1 求解问题需要哪些对象

分析非线性方程求根问题的需求，需要一类称为 `Equation` 的对象来表示非线性方程的概念；此外，还需要一类称为 `Solver` 的对象来求解方程。

由于任意非线性方程的函数 $f(x)$ 都具有一致的形式：参数 `double`，返回 `double`。所以，假设定义一类函数指针来统一表示函数 $f(x)$ ：

```
typedef double (*FunX)(double);
```

针对四种方程求解方法：等间隔搜索法、二分搜索法、牛顿迭代法、弦截法，我们将 `Solver` 定义为抽象类，从 `Solver` 派生出四类具体的方程求解器：`StepSearchSolver`、`BinarySearchSolver`、`NewtonSolver`、`ChordSectionSolver`。

```
class Equation { ... };
```

```

class Solver { ... };
class StepSearchSolver : public Solver { ... }; //等间隔搜索法
class BinarySearchSolver: public Solver { ... }; //二分搜索法
class NewtonSolver: public Solver { ... }; //牛顿迭代法
class ChordSectionSolver: public Solver { ... }; //弦截法

```

2.2 第一层问题：main 函数

如果上述预期的假设成立，则建立对象然后通过对象之间的沟通协作可以实现问题求解，故编写 main 函数易如反掌。尝试编写如下：

```

double f1(double x);
double df1(double x);
int main(){
    Equation equ(f1, "x*x*x - 3.2*x*x + 1.9*x + 0.8");
    int a = -5, b = 5;
    Solver* *pSolver = new Solver*[4];
    pSolver[0] = new StepSearchSolver(equ, a, b);
    pSolver[1] = new BinarySearchSolver(equ, a, b);
    pSolver[2] = new NewtonSolver(equ, df1, a, b);
    pSolver[3] = new ChordSectionSolver(equ, a, b);
    for(int i=0; i<4; i++) {
        pSolver[i]->solve(); pSolver[i]->printSolutions();
    }
    for(int i=0; i<4; i++) { delete pSolver[i]; }
    delete[] pSolver;
    return 0;
}
double f1(double x) { return x*x*x - 3.2*x*x + 1.9*x + 0.8; }
double df1(double x) { return 3*x*x - 6.4*x + 1.9; }

```

在 main 函数中，首先创建了一个 Equation 对象 equ 表示如下方程：

$$x^3 - 3.2x^2 + 1.9x + 0.8 = 0$$

然后，新建了四个具体的方程求解器对象，代表求解方程的四种方法。将四个方程求解器对象的指针保存在一个动态数组中。

接着，for 循环调用方程求解器对象的 solve 方法解出方程 equ 的根，进而调用 Solver 对象的 printSolutions 方法输出方程 equ 的根。

最后，调用 delete 释放掉四个方程求解器对象，并释放掉动态数组。

可以看出，在这里针对 Equation 类的构造函数，Solver 类及其四个具体方程求解器的派生类的构造函数和成员函数做出了一系列假设。

(1) Equation 类的构造函数: Equation(FunX, string);

(2) Solver 类的构造函数: Solver(Equation&, double, double);

(3) Solver 类具有成员函数 solve: 用于求解方程得到区间上的根。

(4) Solver 类具有成员函数 printSolutions: 用于输出方程的根。

(5) StepSearchSolver 类的构造函数:

```
StepSearchSolver(Equation&, double, double);
```

(6) BinarySearchSolver 类的构造函数:

```
BinarySearchSolver(Equation&, double, double);
```

(7) NewtonSolver 类的构造函数:

```
NewtonSolver(Equation&, double, double);
```

(8) ChordSectionSolver 类的构造函数:

```
ChordSectionSolver(Equation&, double, double);
```

于是，问题就归结为这一系列假设的实现。

2.3 第二层问题：定义 Equation 类

分析 main 函数的使用需求和方程函数的实际需求，可以推断出如下结论：

(1) Equation 类应该具有两个数据成员：

✧ 用于保存方程的函数，可以进行函数求值，假设 funX;

✧ 用于保存方程函数的字符串形式，可以输出方程，假设 funString。

(2) Equation 类应该具有三个成员函数：

✧ 用于产生对象的构造函数: Equation(FunX, string);;

✧ 用于输出方程的成员函数，假设 void print() const;

✧ 用于方程函数求值的成员函数，假设 `double f(double x) const`。

根据上述分析推断的结论，尝试实现 `Equation` 类。定义如下：

```
class Equation {
public:
    Equation(FunX fx, string fs) : funX(fx), funString(fs) {}
    void print() const { cout<<funString<<" = 0.0"<<endl; }
    double f(double x) const { return funX(x); }
private:
    FunX funX; //方程的函数指针
    string funString; //方程的字符串形式
};
```

2.4 第二层问题：初步定义 Solver 类

根据需求假设，我们规划的 `Solver` 类是用于求解方程的抽象类，抽取了所有具体方程求解器的公共属性和行为。分析 `Solver` 类的实际需求。

2.4.1 推断 Solver 类的数据成员和成员函数

(1) `Solver` 类必须知道要求解的方程，故设计数据成员 `equation`。

```
const Equation& equation;
```

(2) `Solver` 类必须知道求解的区间，故设计数据成员 `a`、`b` 表示区间上下限。

```
double a, b;
```

(3) `Solver` 类需要保存求出的一系列实根，故设计数据成员 `solutions`。

```
vector<double> solutions;
```

这里，采用 C++ 标准模块库提供的容器 `vector` 来保存实根。容器 `vector` 是一个模板，可以保存任意数据类型的元素，故通过 `vector<double>` 指明容器中可以存放一系列 `double` 元素。因此，`solutions` 是一个 `vector` 容器对象，可以做如下事情：

- ✧ 调用 `push_back` 方法在容器尾部添加元素：`solutions.push_back(1.0);`
- ✧ 调用 `pop_back` 方法删除尾部元素：`solutions.pop_back();`
- ✧ 使用下标运算 `[]` 返回容器的第 `i` 个元素：`solutions[i];`
- ✧ 调用 `back` 方法返回容器尾部元素：`solutions.back();`

✧ 调用 `size` 方法可以返回容器: `solutions.size()`;

关于 `vector` 容器更多的使用方法, 请参考【第 1 章 3.3C++标准模板库】和本书配套的理论教材。

(4) `Solver` 类具有构造函数: `Solver(Equation&, double, double)`;

(5) `Solver` 类具有成员函数 `solve`: 用于求解方程得到给定区间上的根。

(6) `Solver` 类具有成员函数 `printSolutions`: 用于输出方程的根。

2.4.2 Solver 类初步设计

根据上述关于 `Solver` 类的分析和推理, 尝试编写 `Solver` 类如下:

```
class Solver {
public:
    Solver(const Equation &equ, double a, double b)
        : equation(equ), a(a), b(b) {}
    void solve();
    void printSolutions() const;
protected:
    double a, b; //区间上下限
    vector<double> solutions; //保存方程的根
    const Equation& equation; //const&必须在初始化列表中设定初值
};
```

2.4.3 定义 printSolution 成员函数

`Solver` 类的成员函数 `printSolutions`, 用于打印输出方程的根。由于已经假设方程的根保存在 `vector` 容器 `solutions` 中, 故 `printSolutions` 实现很容易。

```
void Solver::printSolutions() const {
    cout<<endl<<solverName()<<endl; //输出求解方法的名称
    cout<<"方程: ";equation.print();
    cout<<"实根["<<a<<","<<b<<"]: ";
    for(int i=0; i<solutions.size(); i++) {
        cout<<solutions[i]<<"\t";
    }
}
```

```
cout<<endl;
}
```

方程有多种求解方法，在打印输出方程根的时候指出方程求解方法很有必要。因此，假设 `Solver` 类具有成员函数 `solverName`，可以得到具体方程求解方法的名称。在定义 `printSolutions` 时调用了 `solverName()` 用于输出求解方法的名称。

但是，抽象类 `Solver` 其实并不知道具体的方程求解方法的名称，无法给出成员函数 `solverName` 的具体定义，因此，在 `Solver` 类中将 `solverName` 成员函数声明为纯虚函数，未来派生出的具体方程求解器必须给出该函数的具体实现。

```
virtual string solverName() const = 0;
```

2.4.4 定义 solve 成员函数

`Solver` 类的成员函数 `solve`，用于求解方程得到给定区间上的根。实际上，具体如何求解方程 `Solver` 类并不知晓，所以，将 `solve` 声明为纯虚成员函数。

```
virtual void solve() = 0;
```

要求在未来派生出的具体方程求解器中给出该函数的具体实现。至此，问题归结为四种具体方程求解器的定义：

- (1) `StepSearchSolver`：等间隔搜索法
- (2) `BinarySearchSolver`：二分搜索法
- (3) `NewtonSolver`：牛顿迭代法
- (4) `ChordSectionSolver`：弦截法

3. 非线性方程求解方法

对于一般函数方程，若 $f(x)$ 在区间 $[a, b]$ 上连续，且 $f(a)f(b) < 0$ ，则方程 $f(x) = 0$ 在区间 $[a, b]$ 上至少有一个实根，称 $[a, b]$ 为一个有根区间。如果有根区间 $[a, b]$ 中有且仅有一个根，则称 $[a, b]$ 为单根区间。

通常求解方程时，先在给定区间 $[a, b]$ 内找到一系列单根区间，保证不会漏掉方程的根，然后针对每个单根区间计算更精细的近似根。常用的方程求解方法有：等间隔搜索法、二分搜索法、牛顿迭代法、弦截法。

3.1 等间隔搜索法

对于方程 $f(x)=0$, $x \in [a, b]$ 。将给定区间 $[a, b]$ 分割为 N 个区间，每个区间步长为 $h=(b-a)/N$ ，从区间的起点 a 开始按步长 h 逐步计算如下函数值：

$$y_0=f(a), y_1=f(a+h), y_2=f(a+2h), \dots, y_N=f(a+Nh)$$

在计算过程中，如果遇到相邻两个函数值 $y_i * y_{i+1} < 0$ ，则一般认为就找到一个单根区间，取该有根区间的中点作为该实根的近似值。重复该过程从 y_0 到 y_N ，就可以找到方程 $f(x)=0$ 的所有单根区间，并取区间中点作为根的近似值。

等间隔搜索法可以比较全面的搜索给定区间范围，找到所有的单根区间，很大程度上可以保证不会遗漏根。但是，为了保证根的精确性，需要将区间分割的更为细小，设定更大的 N ，所以等间隔搜索法的执行效率不高。

3.2 二分搜索法

对于方程 $f(x)=0$ ，假设区间 $[a, b]$ 是一个单根区间且 $f(x)$ 在区间 $[a, b]$ 上连续。则可以通过将区间反复一分为二的办法搜索方程的根。二分搜索的步骤：

- (1) 初始 $x_0=a$, $x_2=b$;
- (2) 计算中间点: $x_1=(x_0+x_2)/2.0$;
- (3) 计算 $f(x_0)*f(x_1)$ 的值并确定根所在区间:
 - (3-1) 如果 $f(x_0)*f(x_1) < 0$ ，说明根落在区间 $[x_0, x_1]$ 上，故取 $x_2=x_1$;
 - (3-2) 如果 $f(x_0)*f(x_1) > 0$ ，说明根落在 $[x_1, x_2]$ 上，故取 $x_0=x_1$;
 - (3-3) 如果 $f(x_0)*f(x_1)=0$ ，说明 x_1 是所求的根，算法终止。

(4) 判断算法是否满足终止条件，若不满足跳转到 (2) 继续执行；否则算法终止返回 x_1 。终止条件：区间范围变得非常小的时候，即 $|x_2-x_0| < \epsilon$ 。

二分搜索法适用于在单根区间中搜索唯一根，速度比等间隔搜索法更快。但是，如果 $[a, b]$ 不是单根区间，则二分搜索会导致遗漏方程的根。

3.3 牛顿迭代法

对于方程 $f(x)=0$ ，假设区间 $[a, b]$ 是一个单根区间且 $f(x)$ 在区间 $[a, b]$ 上连续。

牛顿迭代法将方程的函数做线性化处理，将方程转化为对应的近似方程，然后构造出迭代公式。首先，将函数 $f(x)$ 在 x_i 处进行泰勒展开并保留一阶导数部分：

$$f(x) \approx f(x_i) + f'(x_i)(x - x_i) = 0$$

$$\Rightarrow x = x_i - f(x_i)/f'(x_i)$$

得到牛顿迭代法的迭代式。算法过程如下：

- (1) 初始设定 $x_1, x_{21}=(a+b)/2.0$;
- (2) $x_1=x_2$, 计算 $x_2=x_1-f(x_1)/f'(x_1)$;
- (3) 若 $|x_2-x_1|<\epsilon$, 算法终止返回 x_2 ; 否则跳转到 (2) 继续执行。

牛顿迭代法适用于在单根区间中搜索唯一根，速度比二分搜索法更快。但是，如果 $[a, b]$ 不是单根区间，则牛顿迭代法会导致遗漏方程的根。此外，牛顿迭代法要求函数的一阶导数，有时候很难得到。

3.4 弦截法

弦截法，通过差商替换牛顿迭代法中的一阶导数，得到如下的迭代公式：

$$x_{i+1} = x_i - f(x_i)(x_i - x_{i-1}) / (f(x_i) - f(x_{i-1}))$$

弦截法适用于在单根区间中搜索唯一根，速度比牛顿迭代法要慢，但是比二分搜索快。但是需要提供两个初值，对于非单根区间会导致遗漏方程的根。

4. 第三层问题：派生出具体方程求解器

方程可能存在多个实根，为了保证根不会遗漏，我们首先采用等间隔搜索法找到粗略的单根区间，然后利用二分搜索法、牛顿迭代法、弦截法计算高精度的近似根。因此，每种具体的方程求解器，都需要纳入等间隔搜索法的主框架之下。

4.1 重新设计 Solver 类

根据这种方程求解的框架，有必要改造一下 Solver 类成员函数 solve 的设计。先前认为 Solver 类对于如何求解方程一无所知，故将 solve 声明为纯虚函数：

```
virtual void solve() = 0;
```

然而，现在的 Solver 类知道：所有的方程求解器都先用等间隔搜索法求出所有的单根区间，然后在做精化处理。故重新设计 solve 成员函数如下：

```

void Solver::solve() {
    double h = (b-a)/N;
    double x0 = a, y0 = equation.f(x0);
    for(double x=a+h; x<=b; x+=h) {
        double y1 = equation.f(x);
        if(y0*y1<0.0) { soleSolve(x0, x); } //单根区间精化处理
        y0 = y1; x0 = x;
    }
}

```

这里，将 `solve` 成员函数改造为非虚的成员函数。在函数中，首先计算区间 **N** 等分的步距 $h=(b-a)/N$ ，然后逐次计算每个区间端点的函数值，如果一个区间的两个端点的函数值的乘积 y_0*y_1 小于 0，则认为该区间就是一个单根区间。最后，对每个单根区间调用成员函数 `soleSolve(x0, x)`，在单根区间中求解更为精确的近似根。

当然，对于如何精化处理单根区间，`Solver` 类的确是一无所知的。因此，将成员函数 `soleSolve` 声明为纯虚函数，其具体实现在派生类中提供：

```
virtual void soleSolve(double a, double b) = 0;
```

考虑到需要记住将区间等分的份数 **N**。故在 `Solver` 类中增加数据成员 **N**，用于保存区间等分份数，默认 **N=2000**。相应地需要修改 `Solver` 类的构造函数。

根据上述分析推理的结论，对 `Solver` 类进行重新设计，代码如下：

```

const int nPARTS = 2000; //默认区间等分份数
class Solver {
public:
    Solver(const Equation &equ, double a, double b,
           int n=nPARTS) : equation(equ), a(a), b(b), N(n) {}
    void solve();
    void printSolutions() const;
protected:
    virtual string solverName() const = 0;
    virtual void soleSolve(double a, double b) = 0;
protected:
    double a, b; //区间上下限

```

```
vector<double> solutions;
const Equation& equation;
int N; //区间分割份数
};
```

至此，问题归结为在四种派生的具体方程求解器中，给出 **Solver** 抽象类中两个纯虚成员函数 **solverName** 和 **soleSolve** 的具体实现。

4.2 派生出四种方程求解器

在具体方程求解器中，知道方程求解方法的名称，知道如何对单根区间进行精化。

4.2.1 等间隔搜索法：StepSearchSolver 类

StepSearchSolver 类实现等间隔搜索法，从抽象类 **Solver** 派生，需要实现 **Solver** 类中的纯虚函数 **solverName** 和 **soleSolve**。代码如下：

```
class StepSearchSolver : public Solver {
public:
    StepSearchSolver(Equation &equ, double a, double b,
                    int n=nPARTS) : Solver(equ, a, b, n) {}
    virtual string solverName() const { return "等间隔搜索法"; }
    virtual void soleSolve(double a, double b);
};

void StepSearchSolver::soleSolve(double a, double b) {
    solutions.push_back((a+b)/2.0);
}
```

这里，**soleSolve** 处理单根区间的精化，直接以单根区间的中点作为近似根。

4.2.2 二分搜索法：BinarySearchSolver 类

BinarySearchSolver 类用于实现二分搜索法，从抽象类 **Solver** 派生，实现抽象类 **Solver** 中的纯虚函数 **solverName** 和 **soleSolve**。

```
class BinarySearchSolver : public Solver {
public:
    BinarySearchSolver(Equation &equ, double a, double b,
```

```

        int n=nPARTS) : Solver(equ, a, b, n) {}
    virtual string solverName() const { return "二分搜索法"; }
    virtual void soleSolve(double a, double b);
};

void BinarySearchSolver::soleSolve(double a, double b) {
    double x;
    while(fabs(b-a)>epsilon) {
        x = (a+b)/2.0;
        if(equation.f(a)*equation.f(x)<0) b = x;
        else a = x;
    }
    solutions.push_back(x);
}

```

4.2.3 牛顿迭代法：NewtonSolver 类

NewtonSolver 类用于实现牛顿迭代法，从抽象类 Solver 派生，实现抽象类 Solver 中的纯虚函数 solverName 和 soleSolve。考虑到牛顿迭代法需要函数的一阶导数，故 NewtonSolver 类需增加一个数据成员 df。

```

class NewtonSolver : public Solver {
public:
    NewtonSolver(Equation &equ, FunX df, double a, double b,
        int n=nPARTS) : Solver(equ, a, b, n), df(df) {}
    virtual string solverName() const { return "牛顿迭代法"; }
    virtual void soleSolve(double a, double b);
private:
    FunX df; //牛顿迭代法需要函数的一阶导数
};

void NewtonSolver::soleSolve(double a, double b) {
    double x0, x1 = (a+b)/2.0;
    do {
        x0 = x1; x1 = x0 - equation.f(x0)/df(x0);
    } while(fabs(x0-x1)>epsilon);
}

```

```
solutions.push_back(x1);
}
```

4.2.4 弦截法：ChordSectionSolver 类

ChordSectionSolver 类用于实现弦截法，从抽象类 Solver 派生，实现抽象类 Solver 中的纯虚函数 solverName 和 soleSolve。

```
class ChordSectionSolver : public Solver {
public:
    ChordSectionSolver(Equation &equ, double a, double b,
                      int n=nPARTS) : Solver(equ, a, b, n) {}
    virtual string solverName() const { return "弦截法"; }
    virtual void soleSolve(double a, double b);
};

void ChordSectionSolver::soleSolve(double a, double b) {
    double x0, x1 = a, x2 = b;
    do {
        x0 = x1; x1 = x2;
        x2 = x1 - equation.f(x1) * (x1-x0)
            / (equation.f(x1)-equation.f(x0));
    } while(fabs(x2-x1)>epsilon);
    solutions.push_back(x2);
}
```

至此，问题基本得以解决。细小的读者注意到程序中经常用到 `epsilon`，表示一个很小的数值，定义如下：

```
const double epsilon = 1e-10;
```

5. 完整代码组织结构

```
//源文件：main.cpp
```

```
#include "stdafx.h"
#include "Equation.h"
double f1(double x) { ... }
double df1(double x) { ... }
```

```
int main() { ... }
//头文件: Equation.h
#pragma once
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const double epsilon = 1e-10;
const int nPARTS = 2000;
typedef double (*FunX)(double);
class Equation { ... };
class Solver { ... };
class StepSearchSolver : public Solver { ... };
class BinarySearchSolver : public Solver { ... };
class NewtonSolver : public Solver { ... };
class ChordSectionSolver : public Solver { ... };
//源文件: Equation.cpp
#include "stdafx.h"
#include "Equation.h"
.....//（具体实现代码请参考正文内容组织完成）
```


疯狂实践系列十二：迷你计算器

题目：设计一个简单的计算器类（类型），要求：从键盘读入算式；可以进行加、减、乘、除运算；运算要有优先级；可以带有括号（）；有排错功能，当用户输入错误的算式时提示用户。例如，如果用户输入： $(3 + 4) * 5 - 7$ ，计算结果应为 28。

1. 训练目标

（1）理解自顶向下、分而治之的问题求解思维。根据问题求解需求分析需要哪些对象，推理对象的属性和行为，体验定义类的迭代修正过程。

（2）理解算术表达式的前缀、中缀和后缀形式，掌握表达式求值的原理和过程。

（3）学习 `vector` 容器对象的建立、插入和访问方法，体会使用 `vector` 容器保存表达式的中缀序列和后缀序列的便利性。

（4）了解栈容器的特性，理解栈容器在表达式求值和后缀序列生成中的作用，掌握使用 `vector` 容器模拟栈容器的方法。

（5）理解枚举数据类型的作用及其在表达运算要素时的使用方法。

（6）掌握通过辅助类 `CharHelper` 将众多字符处理函数集中管理的方法。

2. 自顶向下分治

分析问题求解需求，假设存在 `Expression` 类型的对象，用于表达四则算术表达式，能够检验表达式的有效性，能够输出表达式，能够计算表达式的值。

2.1 第一层问题：main 函数

利用 `Expression` 对象，编写 `main` 函数比较容易，代码如下：

```
string exprs[] = {
    "-7 * ( 4 / 3 - 6.25 ) + 9",      //43.4167
    "(5*7 / (-3 --5))*10",           //175.0
    "3+2*(1+2*(-4/(8-6)+7))",         //25.0
    "32*((2-2)+5)/(-15)",             //-10.667
    "1*3*a* /4",                     //无效表达式
}
```

```

    "2.5+ 6/ 3*4 -3+2 *( 4-3 ) - 8" //1.5
};
int main() {
    for(int i=0; i<sizeof(exprs)/sizeof(string); i++) {
        Expression expr(exprs[i]);    cout<<expr;
        double x;
        if(expr.calc(x)) { cout<<"运算结果: "<<x<<endl; }
        else { cout<<"输入表达式无效"<<endl; }
        cout<<endl;
    }
    return 0;
}

```

根据在 main 函数中的使用要求，可以推断出关于 Expression 类的一些结论：

- (1) Expression 类具有构造函数：Expression(string);
- (2) Expression 类具有公有成员函数 calc，用于计算表达式的值。

```
bool calc(double&) const;
```

- (3) Expression 对象需要重载输出运算符<<，以便支持输出表达式。

```
ostream& operator<<(ostream&, const Expression&);
```

问题归结为定义 Expression 类，要求 Expression 类能够产生表达式对象，能够检验并计算表达式的值，能够输出表达式。需要搞清楚表达式求值的原理。

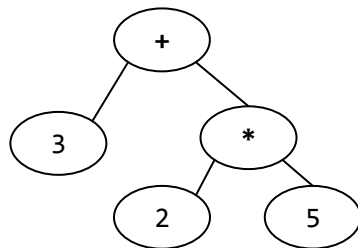
2.2 表达式求值的原理和过程

任意表达式，均可表示成二叉树形式。如“3+2*5”表示成二叉树的形式，如右图所示。图中，叶节点是操作数，除了叶节点之外就是操作符。

在二叉树中，每个非叶节点都有左右两个分支，称为该节点的左子树和右子树，该节点称为中间节点。

对于表达式的二叉树形式，可以采用三种不同的顺序罗列出二叉树的所有节点，称为二叉树的三种遍历方式：

- (1) 中序遍历：按“先左子树，再中间节点，再右子树”的顺序罗列；



(2) 前序遍历：按“先中间节点，再左子树，再右子树”的顺序罗列；

(3) 后序遍历：按“先左子树，再右子树，再中间节点”的顺序罗列；

所谓的中序、前序、后序，其实就是中间节点的访问次序，中间节点最先访问就是前序，中间节点最后访问就是后序，中间节点中间访问就是中序。

采用不同的二叉树遍历方式，同一个表达式可以得到三种不同的表示形式，分别称为表达式的中缀形式、前缀形式和后缀形式。

2.2.1 中缀表达式

以“ $3+2*5$ ”的二叉树表示为例，从根节点“+”开始中序遍历得到中缀表达式。注意，中序遍历的顺序是：先左子树，再中间节点，再右子树。

(1) 首先，罗列“+”节点的左子树，比较简单，罗列结果是：“3”；

(2) 然后，访问中间节点，这里是“+”节点本身，罗列结果是：“3+”；

(3) 最后，罗列“+”节点的右子树。右子树的根节点是“*”节点，

(3-1) 首先，罗列“*”节点的左子树，罗列结果是：“3+2”；

(3-2) 然后，访问中间节点本身“*”，罗列结果是：“3+2*”；

(3-3) 最后，罗列“*”节点的右子树，罗列结果是：“3+2*5”。

最终遍历的结果是“ $3+2*5$ ”，可见我们日常使用的普通表达式，就是中缀表达式。在中缀表达式中，操作符处于操作数的中间位置。

2.2.2 后缀表达式

以“ $3+2*5$ ”的二叉树表示为例，从根节点“+”开始后序遍历得到后缀表达式。注意，后序遍历的顺序是：先左子树，再右子树，再中间节点。

(1) 首先，罗列“+”节点的左子树，比较简单，罗列结果是：“3”；

(2) 然后，罗列“+”节点的右子树。右子树的根节点是“*”节点，

(2-1) 首先，罗列“*”节点的左子树，罗列结果是：“32”；

(2-2) 然后，罗列“*”节点的右子树，罗列结果是：“325”；

(2-3) 最后，访问中间节点本身“*”，罗列结果是：“32 5 *”；

(3) 最后，访问中间节点本身“+”，罗列结果是：“3 2 5 * +”。

最终遍历的结果是"3 2 5 * +", 是表达式 "3+2*5" 的后缀表示形式, 称为后缀表达式。在后缀表达式中, 操作符处于操作数的后面位置。

2.2.3 后缀表达式与计算机求值

中缀表达式, 符合人类的使用习惯, 却不便于计算机求值。因为在中缀表达式顺序扫描过程中, 遇到操作符的时候想执行运算, 但是第二个操作数却没有准备好。这一点恰恰是后缀表达式的优势, 操作数在操作符的前面, 运算时操作数已准备完毕。

后缀表达式的计算求值非常简单。需要借助一种称为栈 (**stack**) 的容器, 这种容器比较特别, 只能在容器的尾部 (栈尾) 插入和删除元素。其中, 在栈尾插入元素的操作称为 **push** (压栈), 在栈尾删除元素的操作称为 **pop** (弹出)。

利用栈进行后缀表达式求值的过程如下:

- (1) 从左向右扫描, 遇到操作数, 就将其 **push** 到栈尾;
- (2) 遇到操作符, 就从栈尾 **pop** 出两个元素做运算, 运算结果 **push** 到栈尾。

以后缀表达式"3 2 5 * +"为例, 从左向右扫描, 遇到 3 压栈, 遇到 2 压栈, 遇到 5 压栈。此时, 扫描到 5, 栈容器的当前状态如下:

后缀表达式

3	2	5	*	+
---	---	---	---	---

栈容器状态

3	2	5
---	---	---

继续向右扫描, 遇到操作符*, 从栈尾弹出 5, 再弹出 2, 执行 $2*5$ 的运算, 结果为 10, 然后把 10 再压栈。此时, 栈容器的当前状态如下:

后缀表达式

3	2	5	*	+
---	---	---	---	---

栈容器状态

3	10
---	----

继续向右扫描, 遇到操作符+, 从栈尾弹出 10, 再弹出 3, 执行 $3+10$ 的运算, 结果为 13, 然后把运算结果 13 压栈。此时的栈容器状态如下:

后缀表达式

3	2	5	*	+
---	---	---	---	---

栈容器状态

13

至此, 表达式从左到右扫描完成, 在栈容器中保留一个数就是最终的计算结果。

2.2.4 表达式求值的过程

根据表达式求值原理, 实现对该表达式的求值需要逐步完成如下几个工作:

(1) 表达式整理：删除原始表达式字符串中非法符号。规定表达式中合法符号包括：操作符 '+' '-' '*' '/'，数字 0~9 和小数点 '.'，括号：'(' ')'。

表达式 "3+2*5"，删除其中的非法符号空格，整理得到："3+2*5"。

(2) 表达式切割：将操作数、操作符和括号分割保存起来。

对整理后的表达式字符串："3+2*5"进行切割，得到一个操作要素的序列，可以将其保存到如下的操作要素容器中，就是中缀表达式序列。

3	+	2	*	5
---	---	---	---	---

(3) 将中缀表达式序列转化为后缀表达式序列。

(4) 对后缀表达式进行求值。

2.3 推断 Expression 类的需求

根据表达式求值原理和过程以及 Expression 对象的使用需求,尝试推断 Expression 类的基本属性和行为能力。

(1) Expression 类应该记住要计算的表达式，故有数据成员 rawExpr。

```
string rawExpr;
```

(2) Expression 类需要把原始的表达式字符串 rawExpr 切割成一个个操作要素，即中缀表达式序列，故定义保存中缀表达式序列的容器，假设叫做 midTokens。

```
vector<Element> midTokens;
```

这里，假设存在 Element 类用于表示操作要素，包括操作符、操作数和括号。

(3) Expression 类应该保存从中缀表达式序列转换来的后缀表达式序列，故定义保存后缀表达式序列的容器，假设叫做 postTokens。

```
vector<Element> postTokens;
```

(4) Expression 类需要产生表达式对象，所以应定义构造函数。

```
Expression(string);
```

(5) Expression 类具有如下公有成员函数，用于计算表达式的值。

```
bool calc(double&) const;
```

(6) Expression 对象需要重载输出运算符<<，以便支持输出表达式。

```
ostream& operator<<(ostream&, const Expression&);
```

(7) Expression 类应该能够剔除表达式字符串中的非法字符。

```
void removeIllegalChar();
```

(8) `Expression` 类应该能够切割表达式字符串，得到中缀表达式序列。

```
void tokenize();
```

(9) `Expression` 类应该能够根据中缀表达式序列生成后缀表达式序列。

```
void toPostExpression();
```

(10) `Expression` 类应该能够从中缀表达式序列生成中缀表达式字符串。

```
string midExpression() const;
```

(11) `Expression` 类应该能够从后缀表达式序列生成后缀表达式字符串。

```
string postExpression() const;
```

3. 第二层问题：定义 `Expression` 类

根据对 `Expression` 类的推断和假设，定义 `Expression` 类，代码如下：

```
class Expression{
public:
    Expression(string);
    bool calc(double&) const;
    string midExpression() const;
    string postExpression() const;
private:
    void removeIllegalChar();
    void tokenize();
    void toPostExpression();
private:
    string rawExpr; //中缀形式的原始表达式
    vector<Element>midTokens;
    vector<Element>postTokens;
};

ostream& operator<<(ostream&, const Expression&);
```

这里，采用 C++ 标准模块库提供的容器 `vector` 来保存中缀表达式和后缀表达式序列。

容器 `vector` 可以保存任意数据类型的元素，通过 `vector<Element>` 指明容器中可以存放一系列操作要素。因此，`midTokens` 和 `postTokens` 容器对象，可以：

- ✧ 调用 `push_back` 方法在容器尾部添加元素：`midTokens.push_back(...);`
- ✧ 调用 `pop_back` 方法删除尾部元素：`midTokens.pop_back();`
- ✧ 调用 `back` 方法访问尾部元素：`midTokens.back();`
- ✧ 使用下标运算`[]`返回容器的第 `i` 个元素：`midTokens[i];`
- ✧ 调用 `size` 方法可以返回容器：`midTokens.size();`

关于 `vector` 容器更多的使用方法，请参考【第 1 章 3.3C++标准模板库】和本书配套的理论教材。

至此，问题归结为 `Expression` 类的成员函数和输出运算符重载函数`<<`的定义。

3.1 重载 `Expression` 的输出运算符

相比较而言，重载 `Expression` 的输出运算符比较简单。代码如下：

```
ostream& operator<<(ostream& os, const Expression& expr) {
    os<<"中缀表达式: "<<expr.midExpression()<<endl;
    os<<"后缀表达式: "<<expr.postExpression()<<endl;
    return os;
}
```

显然，只要 `Expression` 类的成员函数 `midExpression` 和 `postExpression` 存在，则其输出运算符重载函数自然就成立。

幸好，这两个成员函数也很简单。其中，`midExpression` 是把中缀表达式序列的字符串按顺序连接起来，生成中缀表达式字符串；`postExpression` 是把后缀表达式序列的字符串按顺序连接起来，生成后缀表达式字符串。代码如下：

```
string Expression::midExpression() const {
    string expr = "";
    for(int i=0; i<midTokens.size(); i++)
    { expr += midTokens[i].toString() + " "; }
    return expr;
}
string Expression::postExpression() const {
    string expr = "";
```

```
for(int i=0; i<postTokens.size(); i++)
{ expr += postTokens[i].toString() + " "; }
return expr;
}
```

接下来，我们计划解决 `Expression` 类的构造函数，稍微有点艰难。

3.2 定义 `Expression` 类的构造函数

`Expression` 类的构造函数，定义产生 `Expression` 对象的行为。假设给定中缀表达式的字符串，构造函数应该完成如下三个工作：

- (1) 首先，剔除表达式中的非法字符；
- (2) 然后，切割表达式字符串得到中缀表达式序列；
- (3) 最后，根据中缀表达式序列生成后缀表达式序列。

假设 `Expression` 类具有三个成员函数 `removeIllegalChar`、`tokenize` 和 `toPostExpression`，调用这三个成员函数即可完成上述三个工作。代码如下：

```
Expression::Expression(string rexr) : rawExpr(rexr){
    rawExpr = removeIllegalChar(rawExpr);
    tokenize();
    toPostExpression();
}
```

3.2.1 剔除非法字符：定义 `removeIllegalChar`

其中，剔除非法字符的成员函数 `removeIllegalChar` 比较简单，定义如下：

```
void Expression::removeIllegalChar() {
    string legal = "";
    for(int i=0; i<rawExpr.size(); i++) {
        if(CharHelper::isLegal(rawExpr[i])) legal+=rawExpr[i];
    }
    rawExpr = legal;
}
```

在 `removeIllegalChar` 函数中, 假设存在函数 `isLegal` 用于判断表达式字符串中一个字符是否合法。我们规定, 表达式中的合法字符有: 操作符 ('+' '-' '*' '/'), 操作数 (0~9 和小数点), 左右小括号 ('(' ')')。

在后续的编程迭代过程中, 我们会发现这些字符处理函数在很多地方都会重复用到, 如判断是否数字, 是否操作数, 是否操作符, 是否减号, 是否负号, 是否合法等等。为了便于使用和维护, 我们把这些函数集中起来放到一个叫做 `CharHelper` 的辅助类中统一管理, 并且将这些函数全部声明为静态公有成员函数。

```
const char ValidChars[] = {'+', '-', '*', '/', '(', ')', '.' };
class CharHelper {
public:
    static bool isLegal(char ch) {
        if(isDigit(ch)) return true;
        for(int i=0; i<sizeof(ValidChars)/sizeof(char); i++)
            if(ch == ValidChars[i]) return true;
        return false;
    }
    static bool isDigit(char ch) { ... }
    static bool isOperator(char ch) { ... }
    static bool isSubtraction(string str, int i) { ... }
    static bool isNegative(string str, int i) { ... }
    static bool isNum(string str, int i) { ... }
    static string getWholeNum(string str, int& i) { ... }
};
```

静态成员函数是定义在类域内的全局函数, 使用起来非常方便, 不需要产生对象, 直接通过类名限定即可访问, 如 `CharHelper::isLegal(ch)`。

`CharHelper` 辅助类中包含了大量的静态成员函数, 具体定义后续逐渐展开。

3.2.2 生成中缀序列：定义 `tokenize`

`tokenize` 成员函数, 用于切割中缀表达式字符串, 得到中缀表达式序列, 保存到容器 `midTokens` 中。假设存在 `Element` 类用于表示表达式中的操作要素, 且操作要素有 `NUM`、`OPER`、`LEFTBRACKET`、`RIGHTBRACKET` 四种类型。

切割字符串生成中缀序列的处理方式为：从头到尾扫描中缀表达式字符串，

(1) 遇到运算符 ('+' '-' '*' '/'), 则产生一个 **Element** 对象且将其类型设置为 **OPER**, 然后保存到 **midTokens** 容器中。

(2) 遇到左括号, 则产生一个 **Element** 对象且将其类型设置为 **LEFTBRACKET**, 然后保存到 **midTokens** 容器中。

(3) 遇到右括号, 则产生一个 **Element** 对象且将其类型设置为 **RIGHTBRACKET**, 然后保存到 **midTokens** 容器中。

(4) 遇到数字、小数点或者负号, 则获取整个操作数, 然后产生一个 **Element** 对象且将其类型设置为 **NUM**, 然后保存到 **midTokens** 容器中。

最后, 在 **midTokens** 容器中保存的就是中缀表达式序列。

根据上述切割过程, 可以通过注释编写出成员函数 **tokenize** 的代码框架:

```
void Expression::tokenize() {
    for(int i=0; i<rawExpr.size(); ) { //从头到尾扫描
        char ch = rawExpr[i]; //获得第 i 个字符
        if(ch 是数字) {
            //1.1 取出整个操作数, 根据操作数生成 Element 对象;
            //1.2 设定 Element 对象的元素类型为操作数;
            //1.3 将 Element 对象保存到中缀序列 midTokens。
        } else { //如果是操作符
            //2.1 则取出该操作符, 并更加操作符生成 Element 对象
            //2.2 根据操作符的类型, 设定 Element 对象的元素类型
            //2.3 将 Element 对象保存中缀序列 midTokens。
        }
    }
}
```

根据 **tokenize** 的代码框架, 我们需要解决如下几个问题:

- (1) 如何判断“ch 是数字”。注意, 数字字符包括: 0~9, 小数点以及负号。
- (2) 给定操作数的起点位置, 如何从表达式字符串中取出整个操作数。
- (3) 如何产生 **Element** 对象并设定其操作要素的类型。

(4) 如何将 `Element` 对象保存到中缀序列 `midTokens` 中。

其中第(4)个问题非常简单，前文已谈及 `vector` 容器对象的使用方法，通过调用容器的 `push_back` 方法可以在容器尾部添加元素。例如，

```
midTokens.push_back(numElem);
```

第(3)个问题目前有点困难，不过我们可以做出期望的假设。例如，假设 `Element` 类具有如下的构造函数和公有成员函数：

```
//用于产生操作数的 Element 对象
Element(string numString, ELEM_TYPE elemType);
//用于产生操作符和括号的 Element 对象
Element(charoper);
//用于设定操作元素的类型 (NUM、OPER、LEFTBRACKET、RIGHTBRACKET)
void setElemType(ELEM_TYPE etype);
```

当然，还需要假设存在枚举数据类型 `ELEM_TYPE`，用于表达操作要素的类型。

```
enum ELEM_TYPE {NUM, OPER, LEFTBRACKET, RIGHTBRACKET};
```

第(1)和第(2)个问题，我们假设在 `CharHelper` 辅助类中存在两个辅助的成员函数：`isNum` 和 `getWholeNum`。其中，`isNum` 用于判断一个字符是否是操作数的一部分；成员函数 `getWholeNum` 用于将一个完整的操作数从表达式中切割出来。

倘若上述假设成立，定义成员函数 `tokenize` 就比较容易，代码实现如下：

```
void Expression::tokenize() {
    for(int i=0; i<rawExpr.size(); ) {
        if(CharHelper::isNum(rawExpr, i)) {
            string nums = CharHelper::getWholeNum(rawExpr, i);
            Element numElem(nums, NUM);
            midTokens.push_back(numElem);
        } else {
            Element opElem(rawExpr[i]);
            switch(rawExpr[i]) {
                case '(':
                    opElem.setElemType(LEFTBRACKET); break;
                case ')':
                    opElem.setElemType(RIGHTBRACKET); break;
```

```

        case '+': case '-': case '*': case '/':
            opElem.setElemType(OPER); break;
    }
    midTokens.push_back(opElem); i++;
}
}
}

```

3.2.3 完善辅助类 CharHelper

根据前文的假设，CharHelper 辅助类还需要提供 isNum 和 getWholeNum 两个成员函数。其中，isNum 用于判断给定字符是否是操作数的一部分，getWholeNum 用于将一个完整的操作数切割出来。

在判断字符是否是操作数的一部分时，需要区分减法和负号，二者符号相同，但减法是操作符，而负号是操作数。此时，需要根据前后字符做出判别：

- ✧ 如果 '-' 出现在表达式字符串的开头，则必定是负号；
- ✧ 如果 '-' 的前一个字符是其他操作符或者 '('，则必定是负号；
- ✧ 如果 '-' 的前一个字符是 ')' 或者数字，则必定是减号。

因此，进一步假设 CharHelper 辅助类提供 isSubtraction 和 isNegative 两个函数，分别用于判断给定字符是否减法运算符，以及是否负号。

```

class CharHelper {
    static bool isDigit(char ch) {return (ch>='0'&&ch<='9');}
    static bool isOperator(char ch) {
        switch(ch)
        { case '+': case '-': case '*': case '/': return true; }
        return false;
    }
    static bool isLegal(char ch) { ... } //是否合法
    static bool isSubtraction(string str, int i) { //是否减法
        if('-' != str[i]) return false;
        if(i <= 0) return false;
        if(str[i-1] == ')' || isDigit(str[i-1])) return true;
    }
}

```

```
        return false;
    }
    static bool isNegative(string str, int i) { //是否负号
        if('-' == str[i] && !isSubtraction(str, i)) return true;
        return false;
    }
    static bool isNum(string str, int i) {
        if(isDigit(str[i]) || '.' == str[i])
            if(isNegative(str, i)) return true;
        return false;
    }
    static string getWholeNum(string str, int& i) {
        int j=i+1;
        while(j<str.size() && CharHelper::isNum(str, j)) j++;
        string nums(str.begin()+i, str.begin()+j);
        i = j;
        return nums;
    }
};
```

3.2.4 如何生成后缀序列

成员函数 `toPostExpression`，用于从中缀表达式序列生成后缀表达式序列。根据前缀、中缀和后缀表达式的定义，要得到后缀表达式，只需要建立中缀表达式的二叉树表示，然后按后序遍历的顺序遍历二叉树即可得到后缀表达式。

另一种后缀表达式生成方法是利用栈直接转换。本文采用直接转换的办法。

此外还可以借助栈（**stack**）容器进行直接转换。前文已经说明，栈这种容器只能在栈尾插入和删除元素。在栈尾插入元素的操作称为 **push**（压栈），在栈尾删除元素的操作称为 **pop**（弹出）。这里，假设已经建立了一个操作符栈 **opStack**。

利用栈从中缀序列生成后缀序列的过程如下：依次扫描中缀序列中的每个元素，

- （1）如果遇到操作数，将其保存到后缀序列尾部。
- （2）如果遇到操作符，则与 **opStack** 栈顶的操作符比较优先级：

(2-1) 如果栈空或栈顶操作符是左括号，则直接将此运算符入栈；

(2-2) 如果优先级大于栈顶操作符，则该操作符压栈；否则，反复弹出栈顶操作符并将其保存到后缀序列尾部，直到栈顶操作符的优先级低于该操作符的优先级，或者栈空，或者是遇到左括号，最后将该操作符压栈。

(3) 如果遇到左括号，则将左括号压栈。

(4) 如果遇到右括号，则顺序弹出栈顶操作符并将其保存到后缀序列尾部，直到遇到左括号并弹出左括号。

最后，将栈中剩余运算符依次弹出加入后缀序列，得到最终的后缀序列。

以中缀序列"-7 * (4 / 3 - 6.25) + 9"为例，说明生成后缀序列的过程。

(1) 初始栈空。从左向右扫描中缀序列，遇到操作数-7，将其加入后缀序列。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7										
栈状态											

(2) 继续扫描遇到操作符*，与栈顶操作符比较优先级，当前栈空直接将*压栈。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7										
栈状态	*										

(3) 继续扫描，遇到左括号，将左括号压栈。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7										
栈状态	* (

(4) 继续扫描，遇到操作数4加入后缀序列。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7	4									
栈状态	* (

(5) 继续扫描，遇到操作符/，与栈顶的操作符比较优先级，栈顶操作符为*，除法的优先级不大于乘法，故将操作符/压栈。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7	4									
栈状态	*	(/								

(6) 继续扫描，遇到操作数 3 加入后缀序列。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7	4	3								
栈状态	*	(/								

(7) 继续扫描，遇到操作符-，与栈顶的操作符比较优先级，栈顶操作符为/，除法的优先级大于减法，弹出除法操作符并将其保存到后缀序列尾部；继续与栈顶的操作符比较优先级，此时栈顶元素是左括号，故将该运算符压栈。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7	4	3	/							
栈状态	*	(-								

(8) 继续扫描，遇到操作数 6.26 加入后缀序列。继续扫描，遇到右括号。此时需要依次弹出栈顶操作符加入后缀序列，直到遇到左括号并弹出左括号。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7	4	3	/	6.25	-					
栈状态	*										

(9) 继续扫描，遇到操作符+，与栈顶操作符*比较优先级，乘法的优先级大于加法，弹出栈顶操作符*并加入后缀序列。然后栈空，将操作符+压栈。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7	4	3	/	6.25	-	*				
栈状态	+										

(10) 继续扫描，遇到操作数 9 加入后缀序列，扫描结束。

中缀序列	-7	*	(4	/	3	-	6.25)	+	9
后缀序列	-7	4	3	/	6.25	-	*	9	+		
栈状态											

最后将栈中剩余运算符依次弹出加入后缀序列。此时，在后缀序列中保存的就是最终的后缀序列表达式。即：

中缀序列：-7 * (4 / 3 - 6.25) + 9

后缀序列：-7 4 3 / 6.25 - * 9 +

3.2.5 定义 toPostExpression

成员函数 `toPostExpression`，用于从中缀表达式序列生成后缀表达式序列。

已知容器 `midTokens` 中保存了表达式的中缀序列，需要在 `postTokens` 容器中产生表达式的后缀序列。根据后缀序列生成过程，尝试编写代码如下：

```
void Expression::toPostExpression() {
    //1. 定义操作符栈 opStack（可以使用 vector 容器模拟栈容器）
    //2. 扫描中缀序列
    for (int i=0; i<midTokens.size(); i++) {
        Element elem = midTokens[i];
        //2.1 如果 elem 是操作数，则 postTokens.push_back(elem);
        //2.2 如果 elem 是左括号，则 postTokens.push_back(elem);
        //2.3 如果 elem 是操作符，
            //2.3.1 如果栈不空且栈顶不是左括号，且 elem 的优先级
            //大于栈顶操作符，则反复弹出栈顶操作符加入后缀序列
            postTokens.push_back(opStack.back());
            opStack.pop_back();
            //2.3.2 操作符压栈：opStack.push_back(elem);
        //2.4 如果 elem 是右括号，
            //2.4.1 如果栈不空且栈顶不是左括号，则反复弹出
            //栈顶操作符加入后缀序列
            postTokens.push_back(opStack.back());
            opStack.pop_back();
            //2.4.2 弹出左括号：opStack.pop_back();
        }
    //3. 将栈中剩余运算符依次弹出加入后缀序列。
}
```

```

while(!opStack.empty()) {
    postTokens.push_back(opStack.back());
    opStack.pop_back();
}
}

```

在 `toPostExpression` 函数定义中，除了使用 `vector` 容器来保存中缀表达式和后缀表达式序列之外，还通过 `vector` 容器模拟栈容器 `opStack`：

```
vector<Element> opStack;
```

- ✧ 调用 `back` 方法访问栈顶元素：`opStack.back()`;
- ✧ 调用 `push_back` 方法压栈：`postTokens.push_back(...)`;
- ✧ 调用 `pop_back` 方法弹出栈顶元素：`opStack.pop_back()`;
- ✧ 调用 `empty` 方法判断是否栈空：`opStack.empty()`

至此，定义 `toPostExpression` 函数的主要准备工作均已完成。翻译代码如下：

```

void Expression::toPostExpression() {
    vector<Element> opStack;
    for (int i=0; i<midTokens.size(); i++) {
        Element elem = midTokens[i];
        switch(elem.getElemType()) {
            case NUM:
                postTokens.push_back(elem); break;
            case OPER:
                while(!opStack.empty()
                    && !opStack.back().isLeftBracket()
                    && !(opStack.back()<elem)) {
                    postTokens.push_back(opStack.back());
                    opStack.pop_back();
                }
                opStack.push_back(elem); break;
            case LEFTBRACKET:
                opStack.push_back(elem); break;

```

```

        case RIGHTBRACKET:
            while(!opStack.empty() && !opStack.back().isLeftBracket()){
                postTokens.push_back(opStack.back());
                opStack.pop_back();
            }
            opStack.pop_back(); break;
        }
    }
    while(!opStack.empty()) {
        postTokens.push_back(opStack.back());
        opStack.pop_back();
    }
}

```

在 `toPostExpression` 函数及 `Expression` 类定义中，均假设存在 `Element` 类表示一个操作要素（包括操作符、操作数和括号）。现在，是时候兑现承诺了。

3.3 兑现假设：定义 `Element` 类

综合所有关于 `Element` 类的假设，尝试推断 `Element` 类的属性和行为能力：

- (1) `Element` 类具有两个数据成员，分别用于记录操作要素及其类型。

```

string elem;
ELEM_TYPE elemType;

```

这里假设存在枚举数据类型 `ELEM_TYPE`，定义如下：

```

enum ELEM_TYPE {NUM, OPER, LEFTBRACKET, RIGHTBRACKET};

```

- (2) `Element` 类具有两个构造函数，一个用于产生操作数的 `Element` 对象，另一个用于产生操作符和括号的 `Element` 对象。

```

Element(string numString, ELEM_TYPE elemType)
    : elem(str), elemType(etype) { }
Element(char oper) : elem(1, ch) { }

```

- (3) `Element` 类具有 `setElemType` 和 `getElemType` 公有成员函数，用于设定和获取操作元素的类型。

```

void setElemType(ELEM_TYPE etype){ elemType = etype; }

```

```
ELEM_TYPE getElemType() { return elemType; }
```

(4) `Element` 类具有判断操作要素类型的一系列公有成员函数。

```
bool isNumber() const { return elemType==NUM; }
bool isOperator() const { return elemType==OPER; }
bool isLeftBracket() const { return elemType==LEFTBRACKET; }
bool isRightBracket() const { return elemType==RIGHTBRACKET; }
```

(5) `Element` 类应该具有获得操作要素的成员函数，包括获取操作数、获取操作符、获取操作要素的字符串。

```
double toNumber() const {
    assert(isNumber()); double num = DBL_MAX;
    istringstream iss(elem); iss>>num;
    return num;
}
char toOperator() const { return elem[0]; }
string toString() const { return elem; }
```

(6) `Element` 类应该重载小于运算符<，用于比较操作符的优先级。此外，还应该具有获取操作符优先级的成员函数。

```
bool operator<(const Element& elem1) const {
    return priority(elem[0]) < priority(elem1.elem[0]);
}
int priority(char oper) const {
    switch(oper) {
        case '+': case '-': return 1;
        case '*': case '/': return 2;
    }
    return 0;
}
```

至此，万事俱备，剩下的工作就是根据后缀表达式序列进行计算求值了。

3.4 计算求值：定义 `calc` 函数

成员函数 `calc`，用于计算表达式的值，同时检验表达式是否有效。原型如下：

```
bool calc(double&) const;
```

其中，`double&`引用形参用于得到求值结果，`bool` 用于返回表达式是否有效。

已知后缀表达式的求值过程：从左向右扫描，遇到操作数，就将其 `push` 到栈尾；遇到操作符，就从栈尾 `pop` 出两个元素做运算，运算结果 `push` 到栈尾。

3.4.1 编写 `calc` 函数的代码框架

根据后缀表达式求值过程，尝试编写 `calc` 成员函数的代码框架：

```
bool Expression::calc(double &x) const {
    //1. 通过 vector 模拟操作数栈: vector<double> numStack;
    //2. 扫描后缀表达式序列
    for(int i=0; i<postTokens.size(); i++) {
        Element elem = postTokens[i];
        //2.1 若 elem 是操作数则压栈
        //2.2 若 elem 是操作符，则弹出 2 个操作数做运算，运算结果压栈；
        //此时，若栈中元素个数小于 2 则表达式无效，return false;
    }
    //如果操作数栈中元素个数不等于 1，表达式无效，return false;
    //更新 x 为栈顶元素，表达式有效，返回 true
}
```

3.4.2 实现 `calc` 成员函数

根据 `calc` 成员函数的代码框架，给出最终实现。具体代码如下：

```
bool Expression::calc(double &x) const {
    vector<double> numStack;
    for(int i=0; i<postTokens.size(); i++) {
        Element elem = postTokens[i];
        switch(elem.getElemType()) {
            case ELEM_TYPE::NUM:
                numStack.push_back(elem.toNumber()); break;
            case ELEM_TYPE::OPER:
                if(numStack.size()<2) return false;
```

```

        double op2 = numStack.back(); numStack.pop_back();
        double op1 = numStack.back(); numStack.pop_back();
        switch(elem.toOperator()) {
            case '+': numStack.push_back(op1 + op2); break;
            case '-': numStack.push_back(op1 - op2); break;
            case '*': numStack.push_back(op1 * op2); break;
            case '/': numStack.push_back(op1 / op2); break;
        }
        break;
    }
}

if(numStack.size()!=1) return false;
x = numStack.back();
return true;
}

```

4. 完整代码组织结构

以下给出迷你计数器程序的整体代码组织结构，省略部分请根据正文自行组织。

//源文件: main.cpp

```

#include "stdafx.h"
#include "Expression.h"
string exprs[] = { ... };
int main() { ... }

```

//头文件: Expression.h

```

#pragma once
#include "helper.h"
#include "Element.h"
#include <vector>
using std::vector;
class Expression{ ... };
ostream& operator<<(ostream&, const Expression&);

```

//源文件: Expression.cpp

```

#include "stdafx.h"
#include "Expression.h"

```

```
.....
```

```
//头文件: Element.h
```

```
#pragma once
#include <cassert>
#include <string>
#include <sstream>
using std::string;
using std::istringstream;
enum ELEM_TYPE {NUM, OPER, LEFTBRACKET, RIGHTBRACKET};
class Element { ... };
```

```
//源文件: Element.cpp
```

```
#include "stdafx.h"
#include "Element.h"
.....
```

```
//头文件: helper.h
```

```
#pragma once
#include <string>
#include <iostream>
using namespace std;
const char ValidChars[] = {'+', '-', '*', '/', '(', ')', '.' };
class CharHelper { ... };
```


疯狂实践系列之编程无极限

疯狂实践系列至此，已经由浅入深编程求解了十二个稍具规模问题。每个问题，我们都经历了从一无所知，到自顶向下不断层层分治，逐渐获悉问题的面貌和细节，然后大胆假设，谨慎地选择语言语法工具，最终兑现假设的过程。

倘若采用过程化编程思维，面对问题我们就要问自己：解决这个问题需要哪些数据？针对这些数据做哪些加工处理可以实现问题求解。然后，选择相应的语法工具描述数据，描述数据的加工处理过程（包括数据输入输出和数据加工处理）。

倘若采用面向对象编程思维，同样需要回答两个问题：解决问题需要哪些对象，这些对象应该具有哪些属性和行为能力，对象之间如何协作实现问题求解。然后，选择相应的语言语法工具，描述对象类型，描述对象的产生和协作过程。

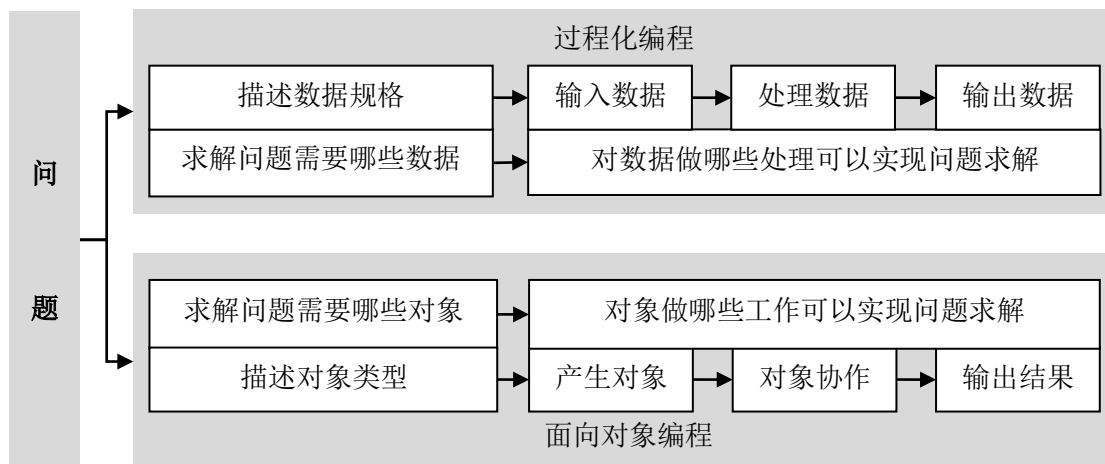


图 35 从问题求解需求到过程化、面向对象编程

无论是过程化还是面向对象编程，在问题求解过程中如果遇到一时难以解决的任务不妨大胆地做出假设，假设存在某个函数能够完成任务，或者存在具有特定属性和行为能力的某种对象能够应付任务，然后以假设成立为前提编程求解问题。进而，在合适的时机兑现假设成立的承诺，给出假设存在的函数或者对象类型的定义。

通过这种自顶向下、层层分治的方法，可以将困难的问题逐渐归结为规模更小、更容易解决的问题，使我们每次面对的问题都相对简单，易于实现，从而能够编写出规模庞大、可靠健壮的程序。必须掌握自顶向下、层层分治的计算思维。

篇幅所限，C++语言疯狂实践系列不可能无限继续下去。但是，学习者的编程实践训练却不能停止。当有一天我们面对实际问题不再无所适从，问题核心需求历历在目，分治方案胸有成竹，语法工具如臂使指，我们高举 C++语言的大旗，指挥着浩浩荡荡的字符大军，横冲直撞，所向披靡，攻克一个又一个的问题堡垒，此时此刻无需强调，编程实践训练已经与我们的生活和工作融为一体，永无休止。

参考文献

- [1] 雷小锋, 毛善君, 张海荣. C++语言探索发现学习教程[M]. 徐州:中国矿业大学出版社, 2015.
- [2] 王晓东. 计算机算法设计与分析[M]. 电子工业出版社, 2007.
- [3] 肖筱南, 赵来军, 党林立. 现代数值计算方法[M]. 北京:北京大学出版社, 2003.
- [4] 微软公司. 使用 Visual Studio 进行调试-Visual Studio 2010 帮助文档.
- [5] 微软公司. 单元测试和 C++-Visual Studio 2010 帮助文档.
- [6] 微软公司. 如何: 创建和运行单元测试-Visual Studio 2010 帮助文档.
- [7] 同济大学数学系. 工程数学线性代数(第六版). 高等教育出版社, 2014.