

C++

第一部分 基础编程

1.1. 初识C++

1.1.1. 编程语言是什么

我们编写程序，就是希望与计算机进行交流，让计算机帮助我们实现我们期望的效果。从这点出发，其实和人与人之间的沟通交流是一样的。两个人如果需要正常的沟通交流，必须要满足的条件是，你说的话对方听得懂，对方说的话你也能听得懂。如果条件不满足，你说中文对方不动，对方说英文你不懂。那么此时就没有办法交流。

写程序也是这样，我们需要和计算机沟通交流。但是很遗憾，机器听不懂我们人类的语言，也学不会人类的语言。那么我们去学习机器的语言呢？更加麻烦，计算机只认识0和1，不会太复杂的语言。此时我们怎么办？

我们可以借助一个“翻译”，把我们的需求翻译成机器语言，说给机器听。但是，这个翻译其实也是一个程序，依然没有办法学习人类的语言。幸运的是，这个“翻译员”有自己的独特的语言体系，而这种语言体系比起机器语言来说，要更加容易理解。这就是我们俗称的“编程语言”。

编程语言有很多很多，有些语言更加贴合机器的世界，这些语言对我们人类来说，学习的成本就比较高。也有一些语言，更加贴合人类的世界，这类的编程语言，更加符合我们人类的生活习惯和思维逻辑，学习起来难度要小很多，更加容易一些。这样的语言，我们称为“高级语言”，而Java就是一种高级语言。

1.1.2. 进制

1.1.2.1. 进制的介绍

在我们正式开始学习C++之前，需要对计算机的一些理论的基础有一定的认知。而进制就是这样的基础，因此我们需要先学习进制。

我们知道在计算机的世界中，只有0和1两个数字，那么其他的数据该如何去表示呢？

1.1.2.2. 什么是进制

进制也就是进位计数制，是人为定义的带进位的计数方法（有不带进位的计数方法，比如原始的结绳计数法，唱票时常用的“正”字计数法，以及类似的tally mark计数）。对于任何一种进制---X进制，就表示每一位置上的数运算时都是逢X进一位。十进制是逢十进一，十六进制是逢十六进一，二进制就是逢二进一，以此类推，x进制就是逢x进位。

1.1.2.3. 进制的分类

二进制

用数字0和1表示每一个自然数，逢2进1。这样的进位制度称为“二进制”

例如：0, 1, 10, 11, 100, 101, 110, 111, 1000

在计算机的世界中，所有的数据最终在存储和运算的时候，都是以二进制的形式进行的。

十进制

日常生活中，我们使用到的进位制度。使用数字0–9表示每一个自然数，逢10进1。这样的进位制度称为“十进制”

例如：0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ...

八进制

使用数字0–7表示每一个自然数，逢8进1。这样的进位制度称为“八进制”

例如：0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, ...

十六进制

使用数字0–9，a–f表示每一个自然数，逢16进1。这样的进位制度称为“十六进制”

例如：0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, 10, 11, 12, 13, ...

TIPS：为什么会有八进制和十六进制

计算机只认识0和1，因此数据的存储和运算都是以二进制的形式完成的。但是，二进制与我们日常生活中使用的十进制相差较远。二进制数字对我们人类来说，可读性会有很大的影响。因此，人们在二进制和十进制的基础上，衍生出来了八进制和十六进制。一方面满足了与二进制之间转换的便捷性，另一方面也带来了可读性的提升。

1.1.2.4. 不同进制的表示方式

进制有着不同的分类，我们常见的有二进制、八进制、十进制、十六进制。因此当我们写一个数字的时候，需要区分到底是什么进制。

- 二进制：通常以**0b**作为开头，例如：0b1001, 0b1100等
- 八进制：通常以**0**作为开头，例如：012, 076, 0234等
- 十六进制：通常以**0x**作为开头，例如：0x12, 0xa2df, 0x23fa等
- 十进制：前面什么都不写，默认的就是十进制的表示形式

TIPS：

在进行二进制的计算的时候，有一个小的技巧：在二进制中，每向左移动一位，相当于在现有的值的基础上乘2。

例如：

0b1 = 1

0b10 = 2

0b100 = 4

0b1000 = 8

基于这一点考虑，我们在进行进制的计算的时候，可以使用拆数字的形式来完成。把一个数字拆解成2的整数次幂，方便累加。

例如：

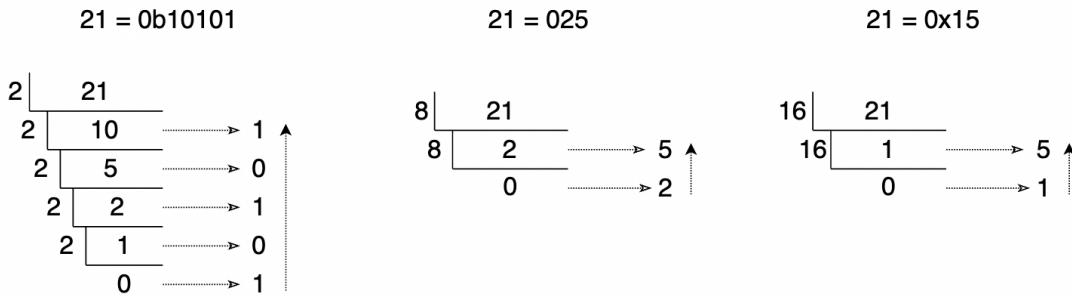
13 = 8 + 4 + 1 = 1101

$$23 = 16 + 4 + 2 + 1 = 10111$$

1.1.2.5. 进制的转换

1. 十进制转其他进制

使用辗转相除法，用数字除进制，再用商除进制，一直累除。直到商为0结束。最后将每一步得到的余数倒着连接起来即可。



2. 其他进制转十进制

用每一位的数字，乘进制的位数-1次方，把所有的结果累加到一起，即可得到十进制表示形式。

$$0b10101 = 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = 21$$

$$021 = 2 \times 8^1 + 1 \times 8^0 = 21$$

$$0x15 = 1 \times 16^1 + 5 \times 16^0 = 21$$

3. 八进制、十六进制与二进制互相转换

- 一个八进制位可以等价替换成3个二进制位
- 一个十六进制位可以等价替换成4个二进制位



```

1 0175 => 0x??
2
3 0b 0111 1101 = 7D
4
5 0x793 => 0?
6 0b 011 110 010 011 = 03623

```

1.1.2.6. 负数的表示方式

在上述的进制的表示和进制的转换中，我们完成的都是正数的表示形式。那么负数呢？

我们需要先知道一个前提：在二进制中，每一个二进制位，我们称为一个比特位，简称bit

8bit = 1Byte (字节)

再往后的单位转换，大家就清楚了：

1024Byte = 1KB

1024KB = 1MB

1024MB = 1GB

1024GB = 1TB

1024TB = 1PB

1024PB = 1EB

...

通常情况下，我们在用二进制表示一个数字的时候，会写满一个字节。如果一个字节表示不了，就用两个字节。如果两个字节表示不了，就用四个字节。如果四个字节表示不了，就用八个字节。以此类推。

因此，数字8的表示形式，就应该是**0000 1000**

在这种表示形式中，最左侧的一位，叫做“最高位”。而这一位不是用来表示大小的，是用来表示正负数的。如果最高位是0表示正数，如果最高位是1表示负数。

```

1 小练习：
2
3 105 = 0110 1001
4 -76 = 1100 1100
5 198 = 0000 0000 1100 0110
6 -82 = 1101 0010

```

1.1.2.7. 原反补

我们已经知道了负数的表示形式了，那么我们进行一些简单的数字的加法试试看：

18 + 12 = ?

18: 0001 0010

12: 0000 1100

和: 0001 1110 = 30

看上去挺好的，结果也都是正常的。那么，我们来做一个减法计算吧：

18 - 12 = ?

18: 0001 0010

12: 0000 1100

差: 0000 0110 = 6

看上去也挺好的，结果也都是正常的。但是，这里是存在问题的。计算机其实并不会做减法计算，只会做加法计算。因此，18-12的计算过程，实际上是 $18+(-12)$ ，我们再来看：

18 + (-12) = ?

18: 0001 0010

-12: 1000 1100

结果: 1001 1110 = -30 ???

结果不对了！

为什么呢？其实原因也很简单，因为符号位是不能直接参与计算的。在刚才的计算中，我们直接用符号位进行计算了，因此得到了错误的结果。

为了避免符号位直接参与计算，导致计算的结果出问题，前人们继续引入了补码的概念。

实际上，在计算机中，所有的数据的存储和计算，都是以补码的形式进行的。

原码：直接计算出来的二进制的表示形式。

反码：正数的反码与原码相同。负数的反码为原码符号位不变，其他位按位取反。

补码：正数的补码与原码相同。负数的补码为反码+1。

1 | 练习：

2 |

3 | -18[补] = 1110 1110

4 | -12[补] = 1111 0100

5 | -76[补] = 1011 0100

那么，我们继续使用补码来进行减法的计算。

```

18 + (-12)

18[补]: 0001 0010
-12[补]: 1111 0100
计算结果: 1 0000 0110

多出来一位怎么办? 简单, 直接舍去就行了! 得到最终的结果: 0000 0110 = 6

```

这样, 计算的结果就正常了! 我们继续做12-18

```

12 + (-18)

12[补]: 0000 1100
-18[补]: 1110 1110
计算结果: 1111 1010

```

不对! 结果好像是一个很大的数字? 因为数据的存储和计算都是以补码的形式进行的, 那么计算的结果也是一个补码。

将补码转成原码其实很简单, 再对这个数字求一次补即可。

```
1111 1010[补] = 1000 0110 = -6
```

结果正常了!

1.1.3. 第一个C++程序

```

1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello World!\n" << std::endl;
5 }

```

1 #include <iostream>	引用指定的库
2	
3 int main() {}	主函数, 程序从这里开始执行
4	
5 std::cout	输出语句, 将后面的内容输出到控制台
6	
7 std::endl	结束输出, 并刷新缓冲区

1.1.4. 注释

在开发的过程中, 我们除了要写我们的程序代码之外, 还有一个非常重要的组成部分是**注释**。很多的公司对注释都是有要求的, 甚至于有的公司会要求注释量不少于代码量。那么, 什么是注释呢? ?

注释其实就是程序员对某一段代码的解释、备注，这些内容是给程序员看的，这部分的内容不会被编译、运行，只是给一段代码做标注。

在C++中，注释分为两种：单行注释，多行注释

1.1.4.1. 单行注释

单行注释以两个斜线 `//` 开始，斜线之后的内容都是注释的内容，不会被编译。

```

1 #include <iostream>
2
3 int main() {
4     // 这里就是单行注释
5     // 下面这段话，可以在控制台上输出 Hello World!
6     std::cout << "Hello World!\n" << std::endl;
7 }
```

1.1.4.2. 多行注释

多行注释以 `/*` 开始，以 `*/` 结束，中间所有的内容都是注释的内容，不会被编译。

```

1 #include <iostream>
2
3 int main() {
4     /*
5         这里的内容就是多行注释的内容
6         可以同时注释多行的内容
7     */
8     std::cout << "Hello World!\n" << std::endl;
9 }
```

1.2. C++数据类型

1.2.1. 标识符

1.2.1.1. 标识符的概念

其实，我们写程序，就是操作各种各样的数据进行计算处理。那么，在程序中的每一个数据，我们需要有一个唯一的标识。这就类似于我们在中学时候学过的代数。

```

1 x = 10
2 y = 20
3 x + y = 30
```

我们在计算 $x+y$ 的时候，知道实际计算的是10和20，也就是说用x代表10，用y代表20。其实，这就是标识符。

标识符，是由字母、数字、下划线组成的一个字符序列，用来表示程序中的数据

1.2.1.2. 标识符的命名规则

- 由字母、数字、下划线组成，不能有其他的组成部分
- 不能以数字开头，需要以字母或者下划线开头
- 不能与系统关键字重复
- 区分大小写

关键字是什么？

其实关键字也是遵循上述前两点规则的字符序列，只不过这样的字符序列已经被系统征用并赋予特殊含义了，我们在定义标识符的时候就不能够再使用了。例如：int、float、const等。

1.2.1.3. 标识符的命名规范

标识符除了在遵循上述的规则之外，还有有一定的规范，并不是可以随便写的。标识符最基础的规范就是望文知义，命名要有一定的意义，方便我们自己去区分标识符所表达的含义。

1.2.2. 数据类型

在程序设计的过程中，我们会操作若干的数据进行各种各样的运算，这些数据由一个个的标识符来表示。但是，每一个数据都是有自己的数据类型的。我们在定义一个变量的时候需要指定数据类型，否则没有办法为这个数据的存储进行空间的开辟。

1.2.2.1. 整型

整型，就是整数的类型，描述的是整数数字。在C++中，整型有如下的一些分类，不同的整型所对应的空间大小不同，能够表示的数据范围也不相同，在适当的场景下，应该选择合适的数据类型来使用。

数据类型	关键字	空间大小	数据范围
短整型	short	2byte	$[-2^{15}, 2^{15} - 1]$
整型	int	4byte	$[-2^{31}, 2^{31} - 1]$
长整型	long	windows: 4byte 非windows 32位: 4byte 非windows 64位: 8byte	$[-2^{31}, 2^{31} - 1]$ $[-2^{31}, 2^{31} - 1]$ $[-2^{63}, 2^{63} - 1]$
长长整型	long long	8byte	$[-2^{63}, 2^{63} - 1]$

```

1 #include <iostream>
2
3 int main() {
4     // sizeof 获取一个数据类型或者一个变量的空间占用大小
5     std::cout << "short: " << sizeof(short) << std::endl;
6     std::cout << "int: " << sizeof(int) << std::endl;
7     std::cout << "long: " << sizeof(long) << std::endl;
8     std::cout << "long long: " << sizeof(long long) << std::endl;
9
10    return 0;
11 }

```

1.2.2.2. 浮点型

浮点型，就是小数的类型，描述的是小数数字。浮点型有两种分类，区别在于占用的空间大小和小数点后面可以精确到的位数。

数据类型	关键字	空间大小	精确范围
单精度浮点型	float	4byte	7位
双精度浮点型	double	8byte	15位

```

1 #include <iostream>
2
3 int main() {
4     // sizeof 获取一个数据类型或者一个变量的空间占用大小
5     std::cout << "float: " << sizeof(float) << std::endl;
6     std::cout << "double: " << sizeof(double) << std::endl;
7
8     return 0;
9 }

```

1.2.2.3. 布尔型

布尔型是用来描述非真即假、非假即真的数据类型，使用关键字 **bool** 来表示，只占用一个字节。布尔型只有两个值true和false。

1.2.2.4. 字符型

字符是用来描述一个文本内容中的最小组成单位的，在程序中使用关键字**char**来表示，只占用一个字节。

1.2.2.5. 字符串型

字符串是由若干个字符组成的一个有序的字符序列，在C++中使用关键字**string**来表示。

1.2.3. 变量、常量

我们设计程序，其实就是需要在内存中开辟出若干空间，存储若干数据，并对这若干的数据进行运算。我们会使用一个标识符来表示这些数据。这些数据中，有些值是可以改变的，例如一个人的年龄、成绩等，这些都是变量。有些值是不可以改变的，比如生日等，这些就是常量。

总结：

可以修改值的是变量，不可以修改值的是常量。

1.2.3.1. 变量

```

1 #include <iostream>
2
3 int main() {
4     // 定义变量的语法：数据类型 标识符
5     int num1;
6     // 也可以在定义变量的同时，设置变量的初始值
7     int num2 = 100;
8     // 变量在定义完成之后，可以修改值
9     num2 = 200;
10
11    // 定义多个相同数据类型的变量
12    int n1 = 100, n2 = 200;
13
14    return 0;
15 }
```

1.2.3.2. 常量

```

1 #include <iostream>
2
3 int main() {
4     // 定义常量的时候，需要添加一个关键字 const
5     const int NUMBER = 10;
6     std::cout << NUMBER << std::endl;
7     return 0;
8 }
```

1.2.3.3. 各种数据类型的变量定义

```
1 #include <iostream>
2
3 int main() {
4
5     // 整型
6     short num1 = 10;
7     int num2 = 20;
8     long num3 = 30;
9     long long num4 = 40;
10
11    // 浮点型
12    float num5 = 3.14f;      // float类型最好在字面量的后面添加f
13    double num6 = 3.14;
14
15    // 布尔型
16    bool num7 = true;
17
18    // 字符型
19    char num8 = 'a';         // 字符需要包含在一对单引号里面
20
21    // 字符串类型
22    string num9 = "hello";   // 字符串需要包含在一对双引号里面
23
24    return 0;
25 }
```

1.2.3.4. 转义字符

我们使用char表示字符类型，将一个字符写到一个单引号中。使用string表示字符串类型的数据，将若干字符写到一个双引号中。

那么，是不是所有的字符都可以写在单引号或者双引号中呢？

并不是！有些字符是具有特殊的含义的。例如，在字符中，单引号成对出现，表示匹配一为字符。在字符串中，双引号成对出现，表示匹配一个字符串。

```

1 // 这种写法是错误的:
2 // 第一个单引号表示一个字符的开始
3 // 第二个单引号匹配一个字符的结束, 此时两个单引号成对, 但是中间并没有字符出现, 已经是错误的了
4 // 第三个单引号就更是错误的情况了
5 char c1 = '''';
6
7 // 这种写法是错误的:
8 // 第一个双引号表示一个字符串的开始
9 // 第二个双引号表示一个字符串的结束, 此时两个双引号成对, 一个字符串结束
10 // 第三个双引号的出现, 就成了错误的情况了
11 string str = """;
```

那么, 我就是想在一个字符中写单引号, 或者在一个字符串中写双引号怎么办? 此时, 就需要使用转义字符了。

转义字符, 就是\, 作用有两个:

- 可以将一个特殊的字符, 变成一个普通的字符。

```

1 #include <iostream>
2
3 int main() {
4     // 在字符中表示单引号
5     char c = '\'';;
6     // 在字符串中表示双引号
7     string str = "\"";
8     return 0;
9 }
```

- 可以配合某些普通的字符, 表达特殊的含义。

```

1 #include <iostream>
2
3 int main() {
4     // n本身是一个普通的字符, 添加上\之后, 就变成了换行符
5     string str1 = "hello \n world";
6
7     // 常见的转义字符配合普通字符的使用:
8     // \n : 换行符
9     // \t : 制表符
10    // \r : 回车符
11    return 0;
12 }
```

1.2.3.5. 数据类型转换

数据类型转换，其实是一个比较奇怪的问题。因为一个变量一旦实例化完成，类型是不能发生改变的。

例如：int a = 10；此时的a从头到销毁，都是int类型，不会改变。

那么，数据类型转换到底是怎么一回事呢？

其实，数据类型转换，指的是定义一个新的指定类型的变量，然后将原来变量的值给这个新的变量进行赋值。这样我们就得到了一个指定数据类型的变量，并且具有了原来变量的值。

数据类型的转换可以分为两种：自动类型转换 和 强制类型转换。

- 自动类型转换

由取值范围小的数据类型，向取值范围大的数据类型转换。转换的过程不需要进行额外的操作，直接赋值就可以了。且转型完成后，不存在值的精度丢失的情况

```

1 #include <iostream>
2
3 int main() {
4     // 1. 定义一个short类型的变量
5     short num1 = 10;
6     // 2. 转型为int类型的变量
7     //    直接赋值即可完成转型
8     int num2 = num1;
9     return 0;
10 }
```

此外，byte、short、char类型的数据参与运算的时候，结果会自动的提升为int类型

- 强制类型转换

由取值范围大的数据类型，向取值范围小的数据类型转换。转换的过程中需要进行强制的类型转换操作，且转型完成后，可能会出现精度丢失的情况。

```

1 #include <iostream>
2
3 int main() {
4     // 1. 定义一个int类型的变量
5     int num1 = 200;
6     // 2. 转型为char类型的变量
7     //    需要进行强制类型转换，在需要转型的变量之前添加小括号，里面写上需要转型的类型即可
8     char num2 = (char)num1;
9
10    // 转型完成后，数据的值可能会发生变化，例如转型为char类型，肯定表示不了200。这种情况叫做
11    // 精度丢失。
12    // 那么，转型之后的结果到底是多少呢？
13    // 前面我们说过数据的存储和运算都是以补码的形式进行的，因此我们来计算一下200的补码是多少
14    // 0000 0000 1100 1000
15    // 转型到char类型的时候，由于只有一个字节，只有八个比特位，因此肯定要舍去一部分的数据。舍
16    // 去的是高位的数据，保留最后的8位
```

```

15 // 也就是说，强制类型转换之后的结果是 1100 1000
16 // 从最高位是1，可以得到是一个负数，那么再计算一下补码，得到1011 1000，也就是-56
17 // 即：上述转型完成后，得到的num2的值是-56
18 return 0;
19 }

```

1.2.3.6. 控制台输入

我们可以使用**cout**将内存中的数据输出到控制台上，也可以从控制台读取数据到内存中，而读取控制台输入的数据，给内存中的某一个变量赋值，需要使用到的是**cin**

```

1 #include <iostream>
2
3 int main(){
4     // 整型变量输入
5     int n = 0;
6     std::cout << "请输入一个整型数值：" << std::endl;
7     std::cin >> n;
8     std::cout << n << std::endl;
9
10    // 浮点型变量输入
11    double f = 0;
12    std::cout << "请输入一个浮点型数值：" << std::endl;
13    std::cin >> f;
14    std::cout << f << std::endl;
15
16    // 字符型变量输入
17    char c = 0;
18    std::cout << "请输入一个字符型数值：" << std::endl;
19    std::cin >> c;
20    std::cout << c << std::endl;
21
22    // 字符串型输入
23    string str;
24    std::cout << "请输入一个字符串型数值：" << std::endl;
25    std::cin >> str;
26    std::cout << str << std::endl;
27
28    // 布尔类型输入
29    bool b = true;
30    std::cout << "请输入布尔型变量：" << std::endl;
31    std::cin >> b;
32    std::cout << b << std::endl;
33
34    return 0;
35 }

```

1.2.3.7. 宏定义

宏定义在C++中是一个比较特殊的命令，它可以在一定程度上方便开发人员的程序设计过程。但是很多的初学者因为不能很好的去理解宏定义，不明白本质是什么，因此在使用宏定义的时候经常会出现问题。

宏定义，就是在文件的头部，使用`#define`来定义一个标识符，用来描述一个字符串。这个标识符就被称为是宏定义。在程序中用到这个宏定义的时候，会直接替换成宏定义描述的字符串。

```

1 #include <iostream>
2
3 #define SUCCESS_CODE 0
4
5 int main() {
6     return SUCCESS_CODE;           // 这里SUCCESS_CODE会被替换成0，也就成了 return 0;
7 }
```

```

1 #include <iostream>
2
3 #define EXPR 2 + 2
4
5 int main() {
6     // 这里的EXPR会被替换成2 + 2，也就成了 std::cout << 2 + 2 << std::endl;
7     std::cout << EXPR << std::endl;
8
9     // 这里的EXPR会被替换成2 + 2，也就成了 std::cout << 2 + 2 * 2 + 2 << std::endl;
10    std::cout << EXPR * EXPR << std::endl;
11
12    return 0;
13 }
```

```

1 #include <iostream>
2
3 #define EXPR (2 + 2)
4
5 int main() {
6     // 这里的EXPR会被替换成(2 + 2)，也就成了 std::cout << (2 + 2) << std::endl;
7     std::cout << EXPR << std::endl;
8
9     // 这里的EXPR会被替换成(2 + 2)，也就成了 std::cout << (2 + 2) * (2 + 2) <<
10    std::endl;
11    std::cout << EXPR * EXPR << std::endl;
12
13    return 0;
14 }
```

1.2.4. 命名空间

在C++中，名称（name）可以是符号常量、变量、函数、结构、枚举、类和对象等等。工程越大，名称互相冲突性的可能性越大。另外使用多个厂商的类库时，也可能导致名称冲突。为了避免，在大规模程序的设计中，以及在程序员使用各种各样的C++库时，这些标识符的命名发生冲突，标准C++引入关键字namespace（命名空间/名字空间/名称空间），可以更好地控制标识符的作用域。

1.2.4.1. 命名空间使用语法

```
1 #include <iostream>
2
3 // 定义一个命名空间
4 namespace A {
5     int num = 10;
6     // 命名空间可以嵌套
7     namespace AA {
8         int num = 30;
9     }
10 }
11 namespace B {
12     int num = 20;
13 }
14
15 // 命名空间是开放的，可以随时把新成员添加到现有的命名空间中
16 namespace A {
17     int score = 100;
18 }
19
20 // 在命名空间中，函数的声明和实现可以分离
21 namespace A {
22     void test();
23 }
24
25 void A::test() {
26     std::cout << "test" << std::endl;
27 }
28
29 int main() {
30     // 输出A命名空间中的num
31     std::cout << A::num << std::endl;
32     // 输出B命名空间中的num
33     std::cout << B::num << std::endl;
34     // 输出A::AA命名空间中的num
35     std::cout << A::AA::num << std::endl;
36     return 0;
37 }
```

1.2.4.2. using

```

1 #include <iostream>
2
3 // 定义命名空间
4 namespace constant {
5     const double PI = 3.141592653;
6     int num1 = 100;
7     int num2 = 200;
8 }
9
10 int main() {
11
12     // 输出PI
13     std::cout << constant::PI << std::endl;
14
15     // using命名空间中的指定成员
16     using constant::num1;
17     std::cout << num1 << std::endl;
18     std::cout << constant::num2 << std::endl;
19
20     // using命名空间
21     using namespace constant;
22     std::cout << num2 << std::endl;
23
24     return 0;
25 }
```

1.2.4.3. 注意事项

```

1 #include <iostream>
2
3 using namespace std;
4
5 // 定义命名空间
6 namespace constant {
7     const double PI = 3.141592653;
8     int num1 = 100;
9     int num2 = 200;
10 }
11
12 namespace A {
13     int num2 = 200;
14 }
15
16 int main() {
17     int num1 = 300;
18     using namespace constant;
```

```

19     cout << num1 << endl;           // 输出当前命名空间中的num1
20     cout << constant::num1 << endl;   // 输出constant命名空间中的num1
21
22
23     using namespace A;
24
25     // 此时, using的constant和A命名空间中都有num2的定义, 且当前的命名空间中没有num2的定义, 此
26     // cout << num2 << endl;
27 }

```

1.3. 运算符

1.3.1. 算术运算符

算术运算符，可以对两个数据进行算术运算。

运算符	含义	示例
+	对两个数字进行相加的计算	10 + 3 = 13
-	对两个数字进行相减的计算	10 - 3 = 7
*	对两个数字进行相乘的计算	10 * 3 = 30
/	对两个数字进行相除的计算	10 / 3 = 3
%	对两个数字进行求模运算（求余数）	10 % 3 = 1

注意事项：

整型与整型的计算结果，还是一个整型。所以，如果 $10/3$ 的时候，得到的结果是 3.3333333 ，此时系统会将这个数字强制类型转换成整型的结果，舍去小数点后面所有的数字，只保留整数部分，也就是3。

在进行计算的时候，结果会进行类型的提升，将结果提升为取值范围大的数据类型。

- int与int的计算结果是int
- int与long的计算结果是long
- float与long的计算结果是float
- float与double的计算结果是double
- ...

```

1 #include <iostream>
2
3 int main(){
4     // 定义两个变量，用来做计算
5     std::cout << 10 + 3 << std::endl;           // 13
6     std::cout << 10 - 3 << std::endl;           // 7
7     std::cout << 10 * 3 << std::endl;           // 30
8     std::cout << 10 / 3 << std::endl;           // 3
9     std::cout << 10 % 3 << std::endl;           // 1
10
11     return 0;
12 }
```

重点来了！自增自减运算符！

`++`、`--`是自增自减运算符，表示在现有的值的基础上，对数据进行`+1`和`-1`的操作。

```

1 #include <iostream>
2
3 int main(){
4     // 定义一个整型的变量
5     int number = 10;
6     // 在现有的值的基础上+1
7     number++;
8     // number现在的结果就是11
9     std::cout << number << std::endl;
10    return 0;
11 }
```

但是！！自增自减运算符，还可以写在变量前面！

- 自增自减运算符，在变量之前，表示先对变量进行计算，再取变量的值使用。
- 自增自减运算符，在变量之后，表示先取变量的值使用，再对变量进行计算。

```

1 #include <iostream>
2
3 int main(){
4     // 定义一个变量
5     int number = 10;
6
7     std::cout << number++ << std::endl;      // 输出10, 结束后number的值自增为11
8     std::cout << ++number << std::endl;      // 输出12, 在上一步结束之后, number的值自增
为了11, 在这里先进行计算, 再取变量的值使用。运算结束后, number的值为12
9     std::cout << --number << std::endl;      // 输出11, 在12的基础上, 先减一, 再输出结果。
运算结束后, number的值为11
10    std::cout << number-- << std::endl;      // 输出11, 在11的基础上, 先取值, 再减一。运算
结束后, number的值为10
11    std::cout << number << std::endl;      // 输出10
12    return 0;
13 }
```

练习题

```

1 // 有一个三位的数字, 计算每一位上的数字的和
2 #include <iostream>
3
4 int main(){
5     // 定义一个三位的数字
6     int number = 123;
7     // 分别获取每一位的数字
8     int g = number % 10;
9     int s = number / 10 % 10;
10    int b = number / 100;
11    // 计算每一位的数字的和
12    int result = g + s + b;
13    std::cout << result << std::endl;
14    return 0;
15 }
```

1.3.2. 赋值运算符

在C++中的赋值运算符是=，可以将等号右边的值，给等号左边的变量进行赋值。

```

1 #include <iostream>
2
3 int main(){
4     // 定义一个变量
5     int variable;
6     // 给这个变量进行赋值
7     variable = 10;
8
9     // 这里的等号，就是一个赋值运算符，将等号右边的值给左边的变量进行赋值。
10    // 赋值完成后，变量variable的值，就是10
11    return 0;
12 }

```

请注意：在C++中，等号是赋值的意思，并不是比较相等的意思。

赋值运算符的意义其实很简单，就是等号右边的值给左边的变量进行赋值。但与此同时，赋值运算符与上述的算术运算符类似，也是有运算的结果的。赋值运算的结果就是赋值完成以后的变量的值！

```

1 #include <iostream>
2
3 int main(){
4     // 定义一个变量
5     int number;
6     // 给变量赋值，并输出赋值结果
7     std::cout << (number = 100) << std::endl;           // 这里的输出结果是100，因为赋值完成
之后的变量number的值就是100
8     return 0;
9 }

```

此外，程序员都可能会偷懒了，在赋值运算符的基础上，又结合了常用的计算，衍生出来了其他的组合的运算符。

常见的组合运算符：`+=` `-=` `*=` `/=` `%=`

以 `+=` 为例，表示将一个变量，在现有的值的基础上进行加法的计算。

```

1 #include <iostream>
2
3 int main(){
4     // 定义一个变量
5     int number = 10;
6     // 让变量在现有的值的基础上+10
7     number += 10;
8
9     std::cout << number << std::endl;           // number的结果就是20
10    return 0;
11 }
12 // 在上述代码中，我们可以将number += 10，理解为 number = (int)(number + 10);

```

```

13 // 对数字number进行加10的计算，并将计算的结果再强制类型转换成变量原来的类型。
14 // 例如：
15 // char c = 'a';
16 // c += 1;
17 // 最后c的结果是b，其实在这里已经完成了结果的强制类型转换。因为字符型的变量在参与运算的时候，结果会
18 // 自动的转型为int类型
// 因为组合运算符内置了强制类型转换的操作，才会使得将计算的结果再转型为char类型

```

1.3.3. 关系运算符

关系运算，就是对两个数字进行大小、相等的比较的运算。常用的关系运算符如下：

符号	描述	示例
==	判断两个数字是否相等	a == b
!=	判断两个数字是否不相等	a != b
>	判断左边的数字是否大于右边的数字	a > b
<	判断左边的数字是否小于右边的数字	a < b
>=	判断左边的数字是否大于等于右边的数字	a >= b
<=	判断右边的数字是否小于等于右边的数字	a <= b

关系运算符比较的是数值类型和字符类型的数据，得到的比较结果一定是bool类型。

```

1 #include <iostream>
2
3 int main(){
4     std::cout << (10 > 5) << std::endl;           // true
5     std::cout << (10 < 5) << std::endl;           // false
6     std::cout << (10 >= 5) << std::endl;          // true
7     std::cout << (10 <= 5) << std::endl;          // false
8     std::cout << (10 == 5) << std::endl;           // false
9     std::cout << (10 != 5) << std::endl;           // true
10
11     return 0;
12 }

```

1.3.4. 逻辑运算符

逻辑运算，是对两个布尔类型的变量进行的逻辑操作。常见的逻辑运算符如下：

符号	描述	示例
&	与运算，两真即为真，任意一个为假，结果即为假	true & false
	或运算，两假即为假，任意一个为真，结果即为真	true false
!	非运算，非真即假，非假即真	!true
^	异或运算，相同为假，不同为真	true ^ false
&&	短路与，左边的结果为假，右边的表达式不参与运算	false && true
	短路或，左边的结果为真，右边的表达式不参与运算	true false

```

1 #include <iostream>
2
3 int main(){
4     std::cout << (true & false) << std::endl;           // false
5     std::cout << (true | false) << std::endl;           // true
6     std::cout << (!true) << std::endl;                  // false
7     std::cout << (true ^ true) << std::endl;           // false
8
9     int num1 = 10;
10    int num2 = 20;
11    bool res = num1++ > 10 && num2++ < 100;
12    std::cout << "res = " << res << ", num1 = " << num1 << ", num2 = " << num2 <<
13    std::endl;
14
15    return 0;
}

```

1.3.5. 位运算符

位运算，是作用于两个整型数字的运算。将参与运算的每一个数字计算出补码，对补码中的每一位进行类似于逻辑运算的操作，1相当于True，0相当于False。

符号	描述	示例
&	位与运算	10 & 20
	位或运算	10 20
^	位异或运算	10 ^ 20
~	按位取反运算	~10
<<	位左移运算	10 << 2
>>	位右移运算	10 >> 2

`~`: 对数字补码的每一位进行取反，包括符号位。

`<<`: 对数字补码的每一位向左移动指定的位数，右侧补0。

`>>`: 对数字补码的每一位向右移动指定的位数，左侧补符号位。

```

1 #include <iostream>
2
3 int main() {
4     std::cout << (10 & 20) << std::endl;           // 位与运算
5     std::cout << (10 | 20) << std::endl;           // 位或运算
6     std::cout << (10 ^ 20) << std::endl;           // 位异或运算
7     std::cout << (~10) << std::endl;             // 按位取反运算
8     std::cout << (10 << 2) << std::endl;           // 位左移运算
9     std::cout << (10 >> 2) << std::endl;          // 位右移运算
10
11    return 0;
12 }
```

1.3.6. 三目运算符

三目运算符是一个带有简单逻辑的运算符，用`?:`来表示，语法如下：`condition ? value1 : value2`

语义：`condition`是一个`bool`类型的变量或者`bool`结果的表达式，如果为`true`，运算的结果为`value1`；如果为`false`，运算的结果为`value2`。

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     // 从控制台输入两个数字，计算两个数字的最大值
7     int num1, num2;
8     cout << "请输入一个数字: ";
9     cin >> num1;
10    cout << "请再次输入一个数字: ";
11    cin >> num2;
12
13    int res = num1 > num2 ? num1 : num2;
14    cout << "最大值是: " << res << endl;
15
16    return 0;
17 }
```

1.4. 流程控制

1.4.1. 流程控制概述

1.4.1.1. 程序的执行结构

我们之前在写代码，执行代码的时候，发现写的代码都是一行一行的执行的。这种执行的结构称为“顺序结构”，除了这种结构之外，还有其他的执行结构。

程序执行结构	结构的描述	描述图
顺序结构	代码从上往下，依次执行。	<pre> graph TD A[A] --> B[B] </pre>
分支结构	程序在某一个节点遇到了多种向下执行的可能性，根据条件，选择一个分支继续执行。	<pre> graph TD P{P} -- T --> A[A] P -- F --> B[B] A --> Join(()) B --> Join Join --> Next </pre>
循环结构	某一段代码需要被重复执行多次。	<pre> graph TD P{P} -- T --> A[A] A --> P P -- F --> Next </pre>

1.4.1.2. 流程控制的概念

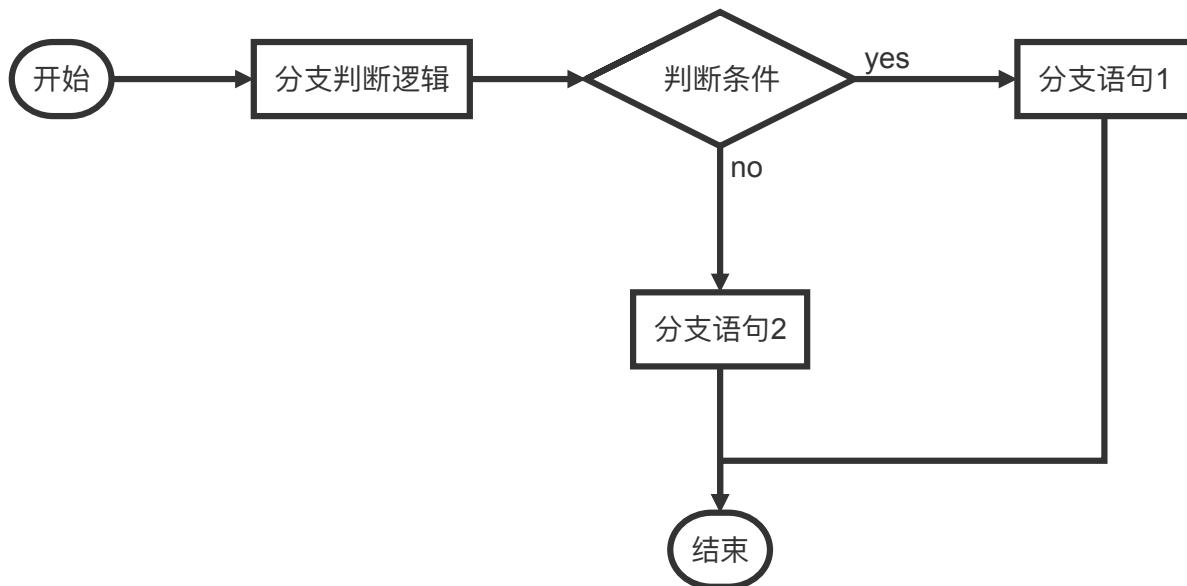
流程控制，就是通过指定的语句，修改程序的执行结构。按照修改的不同的执行结构，流程控制语句可以分为：

- 分支流程控制语句：
 - 顾名思义，就是将程序的执行结构修改为分支结构。
 - 常见的分支流程控制语句有：if-else、switch-case
- 循环流程控制语句：

- 顾名思义，就是将程序的执行结构修改为循环结构。
- 常见的循环流程控制语句有：for、while、do-while

1.4.2. 分支结构

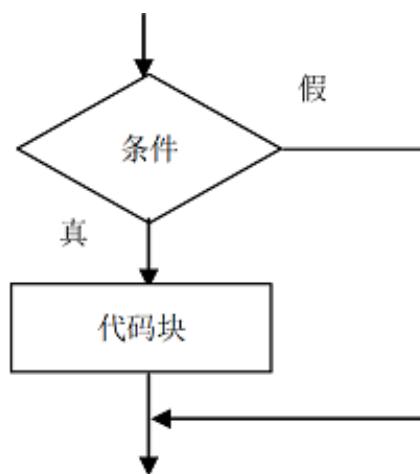
1.4.2.1. if的结构图



1.4.2.2. if的基础语法

```

1 if /* 条件判断 true or false */ {
2     // 条件结果为true执行大括号以内的语句
3 }
4 /*
5   执行流程：
6   代码运行到if分支结构，首先判断if之后的条件是否为true，如果为true，执行大括号里面的语句，如果为
7   false直接执行大括号之外的语句，
8 */
  
```



```

1 // 示例1：成绩如果大于60，奖励一颗糖
2
3 // 简单的if语句：
4 // 成绩如果大于60    给奖励
5 int score = 10;
6 if (score>60) {
7     cout << "给颗糖" << endl;
8 }

```

```

1 // 示例2：C++成绩大于98分，而且MySQL成绩大于80分，老师奖励他；或者C++成绩等于100分，MySQL成绩大于70分，老师也可以奖励他。
2
3 if ((cppScore >98 && mysqlScore > 80 ) || ( cppScore == 100 && mysqlScore > 70 )) {
4     //奖励
5 }

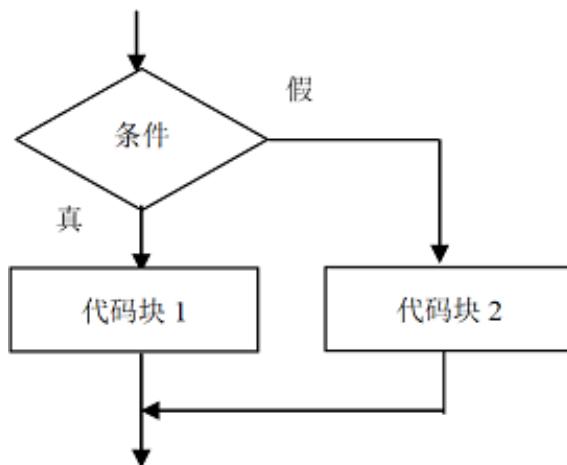
```

1.4.2.3. if-else的使用

```

1 if (condition) {
2     // 代码段1
3 }
4 else {
5     // 代码段2
6 }
7 // 逻辑： condition是一个boolean类型的变量，或者一个boolean结果的表达式。如果condition的值为 true，则代码段1执行，否则，代码段2执行

```



```

1 int score;
2 cout << "请输入一个成绩:" ;
3 cin >> score;
4 if (score >= 60) {
5     cout << "成绩及格了! " << endl;
6 }
7 else {
8     cout << "成绩不及格! " << endl;
9 }
```

- 示例1：如果是男生就永远18岁，否则永远16岁。

```

1 string gender;
2 cout << "请输入你的性别: " ;
3 cin >> gender;
4 if (gender == "男") {
5     cout << "永远18岁! " << endl;
6 } else {
7     cout << "永远16岁! " << endl;
8 }
```

- 示例2：买彩票，如果体彩中了500万，我买车、买房、非洲旅游，如果没中，继续买。

```

1 bool isWin;
2 cout << "你中彩票了吗? " ;
3 cin >> isWin;
4 if (isWin) {
5     cout << "买车! 买房! 非洲旅游! " << endl;
6 } else {
7     cout << "继续搬砖" << endl;
8 }
```

1.4.2.4. else if的使用

```

1 if (condition1) {
2     // 代码段1
3 }
4 else if (condition2) {
5     // 代码段2
6 }
7 else {
8     // 代码段3
9 }
10 // 逻辑：先判断condition1，如果condition1成立，执行代码段1；如果condition1不成立，再判断
    condition2，如果condition2成立，执行代码段2，否则执行代码段3
```

- 示例1：在控制台输入成绩，输出成绩的等级

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int score;
6     cout << "请输入你的成绩:" ;
7     cin >> score;
8     if (score < 60) {
9         cout << "不及格" << endl;
10    } else if (score < 80) {
11        cout << "良好" << endl;
12    } else {
13        cout << "优秀!" << endl;
14    }
15
16    return 0;
17 }
```

- 如果成绩大于90并且是男生就送个女朋友，成绩大于90并且是女生送个男朋友，否则...

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int score;
6     cout << "请输入你的成绩:" ;
7     cin >> score;
8
9     string gender;
10    cout << "请输入你的性别:" ;
11    cin >> gender;
12
13    if (score > 90 && gender == "男") {
14        cout << "送你一个女朋友" << endl;
15    } else if (score > 90 && gender == "女") {
16        cout << "送你一个男朋友" << endl;
17    } else {
18        cout << "洗洗睡吧" << endl;
19    }
20
21    return 0;
22 }
```

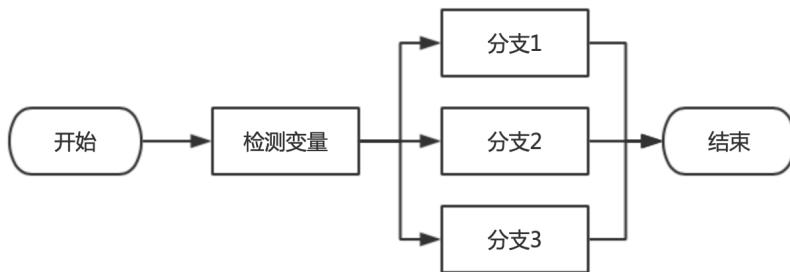
- 在控制台上输入一个字母，如果是小写字母，将其转成大写输出；如果是大写字母，将其转成小写输出；如果不是字母，原义输出。

```
1 #include <iostream>
```

```

2
3 using namespace std;
4
5 int main() {
6
7     char c ;
8     cout << "请输入一个字符";
9     cin >> c;
10    if (c >= 'a' && c <= 'z') {
11        cout << (char)(c - 32) << endl;
12    } else if (c >= 'A' && c <= 'Z') {
13        cout << (char)(c + 32) << endl;
14    } else {
15        cout << c << endl;
16    }
17
18    return 0;
19 }
```

1.4.2.5. switch结构图



1.4.2.6. switch基础语法

```

1 switch(variable) {
2     case const1:
3         statement1;
4         break;
5     case const2:
6         statement2;
7         break;
8         ...
9     case constN:
10        statementN;
11        break;
12    default:
13        statement_default;
14        break;
15 }
```

程序逻辑：

- 检测某一个变量的值，从上往下依次与每一个case进行校验、匹配
- 如果变量的值和某一个case后面的值相同，则执行这个case后的语句
- 如果变量的值和每一个case都不相同，则执行default后的语句

1.4.2.7. switch的语法规则

1. 表达式expr的值必须是整型
2. case子句中的值const，必须是常量值，case中的值不能是一个范围
3. 所有case子句中的值应是不同的，否则会编译出错
4. default子句是可选的（不是必须的）
5. break语句用来在执行完一个case分支后使程序跳出switch语句块；否则会继续执行下去

1.4.2.8. switch基础案例

```

1 // 简单实现switch-case语句
2 int i = 1;
3 switch(i){
4     case 1:
5         cout << "Hello World!" << endl;
6         break;
7     case 2:
8         cout << "Hello World!2" << endl;
9     case 3:
10        cout << "Hello World3" << endl;
11        break;
12    default:
13        cout << "Haaaa" << endl;
14        break;
15 }
```

```

1 // 判断春夏秋冬
2 int season = 1;
3 switch(season){
4     case 1:
5         cout << "春暖花开" << endl;
6         break;
7     case 2:
8         cout << "闷热" << endl;
9         break;
10    case 3:
11        cout << "秋高气爽" << endl;
12        break;
13    case 4:
14        cout << "滴水成冰" << endl;
15        break;
```

```

16     default:
17         cout << "火星的" << endl;
18         break;
19 }
```

1.4.2.9. switch的穿透性

我们来看一段代码

```

1 // 判断春夏秋冬
2 int season = 1;
3 switch(season){
4     case 1:
5         cout << "春暖花开" << endl;
6     case 2:
7         cout << "闷热" << endl;
8     case 3:
9         cout << "秋高气爽" << endl;
10    case 4:
11        cout << "滴水成冰" << endl;
12    default:
13        cout << "火星的" << endl;
14 }
```

上述代码中，switch结构捕获变量season的值。变量的值和第一个case是匹配的，应该输出的结果是“春暖花开”。但实际上的输出结果却是从“春暖花开”开始的每一个输出。

因为在switch结构中有“穿透性”。

穿透性：

指的是，当switch的变量和某一个case值匹配上之后，将会跳过后续的case或者default的匹配，直接向后穿透。

```
// 判断春夏秋冬  
int season = 1;  
switch(season){  
    case 1:  
        cout << "春暖花开" << endl;  
    case 2:  
        cout << "闷热" << endl;  
    case 3:  
        cout << "秋高气爽" << endl;  
    case 4:  
        cout << "滴水成冰" << endl;  
    default:  
        cout << "火星的" << endl;  
}
```



为了杜绝穿透， 可以使用关键字 `break`:

```
// 判断春夏秋冬
int season = 1;
switch(season){
    case 1:
        cout << "春暖花开" << endl;
    case 2:
        cout << "闷热" << endl; 
        break;
    case 3:
        cout << "秋高气爽" << endl;
        break;
    case 4:
        cout << "滴水成冰" << endl;
        break;
    default:
        cout << "火星的" << endl;
        break;
}
```

1.4.2.10. 合理的使用穿透性

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     // 输入一个年月日，判断这一天是这一年的第几天
7     int year, month, day;
8     cout << "请输入一个年:" ;
9     cin >> year;
10    cout << "请输入一个月:" ;
11    cin >> month;
12    cout << "请输入一个日:" ;
13    cin >> day;
14
15    // 定义变量，记录总天数
16    int days = day;
17
18    switch (month) {
19        case 12: days += 30;
20        case 11: days += 31;
21        case 10: days += 30;
22        case 9: days += 31;
```

```

23     case 8: days += 31;
24     case 7: days += 30;
25     case 6: days += 31;
26     case 5: days += 30;
27     case 4: days += 31;
28     case 3: days += year % 4 == 0 && year % 100 != 0 || year % 400 == 0 ? 29 :
29         case 2: days += 31;
30     }
31
32     cout << days << endl;
33
34     return 0;
35 }
```

1.4.2. 循环结构

1.4.2.1. while循环的基础语法

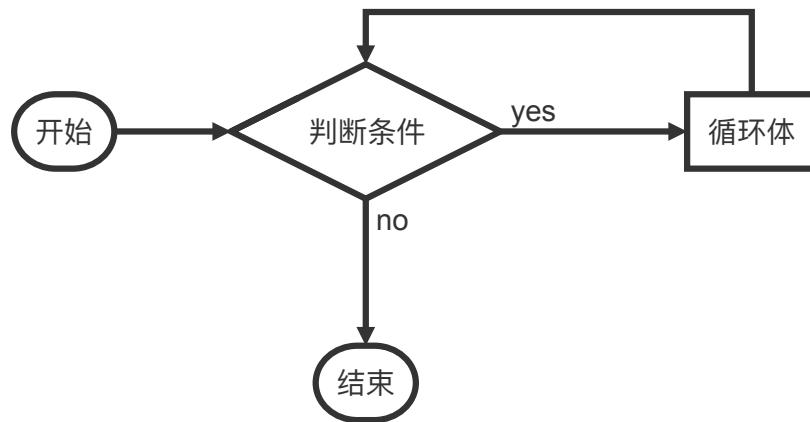
```

1 // while语句由关键字 while 小括号 大括号 以及相关语句组成
2 // 基本格式如下:
3 while (条件表达式) {
4     循环体
5 }
```

条件表达式：循环终止的判断条件语句，要求为布尔表达式，也就是结果为真或假值 比如 `i < 10;`

循环体：n行循环要执行的语句

1.4.2.2. while的结构



流程说明：

1. 执行条件表达式，也就是执行循环终止的判断条件语句，看其结果是true还是false，如果是 false，循环结束，如果是true继续执行
2. 执行循环体语句，也就是执行大括号中的实际代码

3. 回到第一步再次执行，第一到第三步，直到某次第一步表达式结果为false

1.4.2.3. while循环的注意事项

1. while循环本身没有循环变量声明和初始化的组成部份，所以如果有这个需要，应在while循环前声明循环变量并赋值
2. while循环本身也没有控制循环终止的判断条件语句部分，所以需要在循环体中增加相应的控制语句，否则容易死循环

1.4.2.4. while循环案例

1. 打印输出5次，我爱编程，我爱千锋

```

1 // 初始化部分
2 int count = 0;
3 // 循环条件
4 while(count < 5){
5     // 循环体
6     cout << "我爱编程，我爱千锋" << endl;
7     // 更新循环变量
8     count++;
9 }
```

2. 求 10 的阶乘

```

1 int sum = 1;
2 int i = 1;
3 while(i <= 10){
4     sum = sum * i;
5     i++;
6 }
7 cout << "10的阶乘：" << sum << endl;
```

3. 求1-100的和

```

1 int i = 1;          // 初始化变量
2 int sum = 0;        // 保存和
3 // 循环条件
4 while(i <= 100){
5     sum = sum + i;
6     i++;
7 }
8 cout << "1-100的和是：" << sum << endl;
```

4. 求100以内的偶数的和

```

1 int z = 2;
2 int sum = 0;
3 while(z <= 100){
4     sum = sum + z;
5     z += 2;
6 }
7 cout << "1-100的偶数的和是: " << sum << endl;
8
9 // 或
10 int z = 1;
11 int sum = 0;
12 while(z <= 100){
13     if(z % 2 == 0){
14         sum = sum + z;
15     }
16     z++;
17 }
18 cout << "1-100的偶数的和是: " << sum << endl;

```

1.4.2.5. do-while循环的基本语法

```

1 // do-while语句由 do关键字 大括号 while关键字 小括号号和相关语句组成
2 // 基本格式如下:
3 do {
4     循环体
5 } while (条件表达式);

```

1.4.2.6. do-while循环的执行结构

1. 执行循环体语句，也就是执行大括号中的实际代码
2. 执行条件表达式，也就是执行循环终止的判断条件语句，看其结果是true还是false，如果是 false，循环结束，如果是true 继续执行
3. 回到第一步再次执行第一到第三步，直到某次第一步表达式结果为false

1.4.2.7. do-while循环的特点和注意事项

1. do-while循环为先执行后判断，也就是先执行一次循环体中的代码，然后再检查条件表达式，所以do-while循环至少会执行一次

2. 其它特点和while循环一样

1.4.2.8. do-while循环练习

1. 打印三次helloworld

```

1 // 1 初始化部分
2 int i = 0;
3 do{
4     // 2 循环体
5     cout << "Hello World!" << endl;
6     // 4 循环变量变化部分
7     i++;
8 }while(i < 3); // 3 循环条件

```

2. 用do/while实现打印100以内的奇数

```

1 int j = 1;
2 do{
3     cout << j << endl;
4     j += 2;
5 } while (j < 100);

```

3. 100 以内能够被3整除，但是不能被5整除的数打印输出

```

1 int z = 3;
2 do{
3     if(z % 3 == 0 && z % 5 != 0){
4         cout << z << endl;
5     }
6     z++;
7 }while(z<=100);

```

1.4.2.9. for循环的基础语法

```

1 // for语句由关键字for、小括号、大括号以及相关语句组成
2 // 基本格式如下
3 for(循环起点； 循环条件； 循环步长) {
4     循环体
5 }

```

循环起点： 循环变量初始化语句，比如 `int i = 0;`

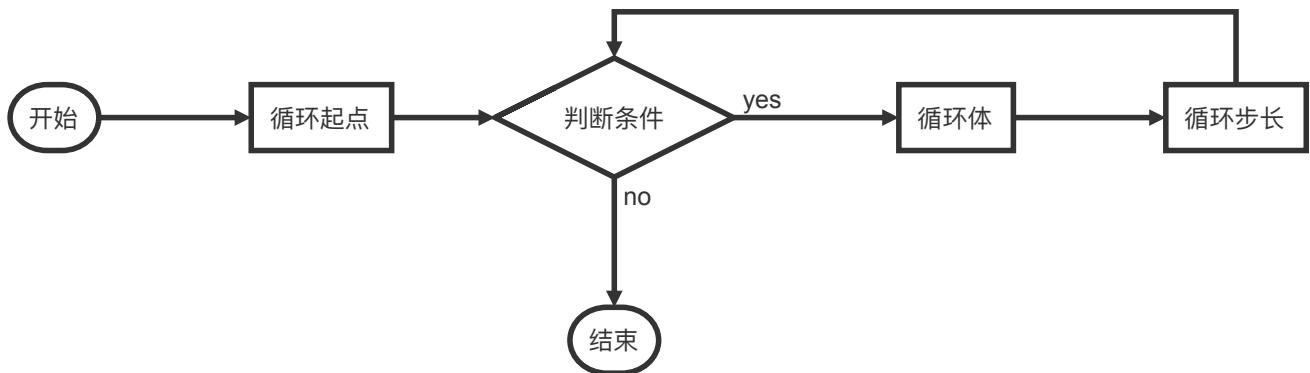
循环条件： 循环终止的判断条件语句，要求为布尔表达式，也就是结果为真或假值 比如 `i < 10;`

循环步长： 循环改变的控制条件语句，比如 `i++`

循环体：循环要执行的语句

三个表达式之间用分号分隔

1.4.2.10. for循环的结构图



流程说明：

1. 执行循环起点，也就是执行循环变量初始化语句
2. 执行判断条件，也就是执行循环终止的判断条件语句，看其结果是true还是false
如果是 false，循环结束。
如果是 true，继续执行第三步。
3. 第三步：执行循环体语句，也就是执行大括号中的实际代码
4. 第四步：执行表达式3，也就是执行循环改变的控制条件语句，使循环变量的值发生改变
5. 第五步：回到第二步再次执行第二到第五步，直到第二步的表达式结果为false，循环结束

```

1 //实操代码
2 for(int i = 0; i < 10; i++){
3     cout << "当前循环变量的值是" << i << endl;
4 }
5 /*
6 第一步：执行 int i=0;
7 第二步：执行 i<10; (此时 i值为0) 结果为true,所以程序继续执行第三步
8 第三步：执行 cout << "当前循环变量的值是" << i << endl; 打印输出
9 第四步：执行 i++ (执行后 i值为 1)
10 第五步：执行 (回到第二步执行) i<10; (此时 i值为1) 表达式结果依然为 true,所以程序继续 (执行三到五步，直到某次第二步的语句结果为false ,程序结束)
11 */
  
```

1.4.2.11. for循环基础案例

1. 打印输出3次helloworld

```

1 | for(int i = 0; i < 3; i++) {
2 |     cout << "Hello World!" << endl;
3 |

```

2. 打印100以内，能被4整除不到能被7整除的数据，每行打印6个

```

1 | int count = 0;
2 | for(int i = 1; i <= 100; i++) {
3 |     if(i % 4 == 0 && i % 7 != 0){
4 |         cout << i << "\t";
5 |         count++;
6 |         if(count % 6 == 0) {
7 |             cout << "\n";
8 |         }
9 |     }
10 |

```

3. 打印1到100的所有整数

```

1 | //注意 正常循环变量初值一般为0，这里从1开始，结束条件是否包括等于，需要根据实际需求决定
2 | for(int i = 1 ; i <= 100 ; i++){
3 |     cout << "整数值：" << i << endl;
4 |

```

4. 计算1+2+3+...+100的和

```

1 | //注意 正常循环变量初值一般为0，这里从1开始，结束条件是否包括等于，需要根据实际需求决定
2 | int sum = 0;
3 | for(int i = 1 ; i <= 100 ; i++){
4 |     sum += i;
5 | }
6 | cout << "1到100的所有整数和为：" << sum << endl;

```

5. 统计1到100之间分别能被3整除或5整除或同时被3和5整除的数字个数并打印

```

1 | #include <iostream>
2 | #include <sstream>
3 |
4 | using namespace std;

```

```

5
6 int main() {
7
8     // 记录能被3整除的数字个数、能被5整除的数字个数、能被3和5整除的数字个数
9     int count3 = 0, count5 = 0, count35 = 0;
10    // 使用字符串流记录能被3整除的数字、能被5整除的数字、能被3和5整除的数字
11    ostringstream oss3, oss5, oss35;
12
13    // 循环遍历数字
14    for (int n = 1; n <= 100; n++) {
15        if (n % 3 == 0 && n % 5 == 0) {
16            count35++;
17            oss35 << n << ", ";
18        } else if (n % 3 == 0) {
19            count3++;
20            oss3 << n << ", ";
21        } else if (n % 5 == 0) {
22            count5++;
23            oss5 << n << ", ";
24        }
25    }
26
27    cout << "能被3整除，不能被5整除的数字有" << count3 << "个" << endl;
28    cout << "分别是：" << oss3.str() << endl;
29
30    cout << "能被5整除，不能被3整除的数字有" << count5 << "个" << endl;
31    cout << "分别是：" << oss5.str() << endl;
32
33    cout << "能同时被3和5整除的数字有" << count35 << "个" << endl;
34    cout << "分别是：" << oss35.str() << endl;
35
36    return 0;
37 }
```

6. 打印九九乘法表

```

1 // 九九乘法表 共有45个结果，所以需要循环45次 第一行输出一个结果，第二行输出二个结果，...第九
行输出九个结果
2 // 用来记录当前行号
3 int row = 1;
4 // 用来记录当前列号（也就是当前行的第几个结果）
5 int col = 1;
6 for (int i = 0; i < 45 ; i++) {
7     //输出row行的第col个结果，不换行
8     cout << col << " x " << row << " = " << (col*row);
9     //同一行中多个结果之间的分隔符
10    cout << "\t";
11    //如果行号和列号相等，说明第row行已打印完成
```

```

12     if(row == col){
13         //打印换行符
14         cout << endl;
15         //列号重置
16         col = 1;
17         //行号加1
18         row++;
19     }
20     else{
21         //列号加1
22         col++;
23     }
24 }
```

7. 判断指定的数字是否是质数并输出

```

1 // 用来记录是否是质数的布尔变量 false 就是质数
2 bool flag = false;
3 // 需要判断的数字，实际可由用户通过键盘输入
4 int checkNumber = 100;
5 // 循环判断当前数字是否能被1和它本身外的数字整除
6 for(int i = 2; i < checkNumber; i++){
7     // 如果能被整除，说明不是质数
8     if(checkNumber % i == 0){
9         // 设置标质不是质数
10        flag = true;
11    }
12 }
13 //根据标记变量的真假，直接输出结果
14 cout << "整数" << checkNumber << (flag?"不是质数":"是质数") << endl;
```

1.4.2.12. for循环的特殊用法

```

1 //第一种，将表达式一省略，放到循环前面
2 int i = 0;
3 for(; i < 10; i++){
4     cout << "当前循环变量的值是" << i << endl;
5 }
6 //第二种，将表达式3省略，放到循环体中
7 int i = 0;
8 for(; i < 10;){
9     cout << "当前循环变量的值是" << i << endl;
10    i++;
11 }
12 //第三种，将表达式1, 2, 3全部省略 此种为死循环，一般需要在循环体中给出结束条件
13 for( ; ; ){
14     cout << "当前循环变量的值是" << i << endl;
15 }
```

1.4.2.13. 流程控制关键字break

- 1 | 1. 作用：break语句用于终止某个语句块的执行
 2 | 2. 用法：如果是循环中，作用为跳出所在的循环，如果是在switch语句中，则为跳出所在的switch语句

跳出break语句所在的循环

```
1 | for (int i = 1; i < 3 ; i++ ) {
2 |     for (int j = 1; j< 5 ;j++ ) {
3 |         if(j == 2){
4 |             break;// 跳出内层循环，进行下一次外层循环
5 |         }
6 |         cout << i << " " << j << endl;
7 |     }
8 | }
```

1.4.2.14. 流程控制关键字continue

- 1 | 1. 作用：跳过本次循环，执行下一次循环（如果有多重循环，默认继续执行离自己最近的循环）提前终止本次循环
 2 | 2. 使用：只能在循环结构中使用

当 j==2时，跳过本次循环，不进行打印输出

```
1 | for (int j = 1;j<4 ;j++ ){
2 |     if(j==2){
3 |         continue;
4 |     }
5 |     System.out.println("i="+i + " j="+j);
6 | }
7 | System.out.println("Hello World!");
```

1.4.2.15. 流程控制关键字goto

- 1 | 在程序中，可以任意的设置“标签”，使用关键字goto可以直接跳转到指定的标签位置继续执行程序！

```

1 label1:
2 cout << "1" << endl;
3 goto label3;
4
5 label2:
6 cout << "2" << endl;
7
8 label3:
9 cout << "3" << endl;

```

```

1 for (int i = 0; i < 10; i++) {
2     for (int j = 0; j < i; j++) {
3         cout << "i = " << i << ", j = " << j << endl;
4         if (i == 4 && j == 3) {
5             goto label1;
6         }
7     }
8 }
9 label1:
10 cout << "end" << endl;

```

1.4.2.15. 多重循环

- 概念：多重循环就是指 在循环内嵌套其它循环，和选择语句嵌套类似，嵌套几层就是几重循环，最常见的为双重和三重
- 说明：外层循环执行一次，它的内层循环执行一轮（也就是内循环正常循环一遍结束）

```

1 第一步：从外层循环开始执行 先执行一次进入循环体（里面包括内层循环）
2 第二步：遇到内层循环，开始正常执行，直到内循环循环结束
3 第三步：继续外循环的下一次循环（回到第一步）

```

1. 用双重循环实现九九乘法表的打印

```

1 for (int i = 1; i <=9 ; i++) {
2     for (int j = 1; j <=i ; j++) {
3         cout << j << "x" << i << "=" << (i*j) << "\t";
4     }
5     cout << endl;
6 }

```

2. 打印直角三角形

```

1 | for (int i = 1;i<=5 ;i++ ){
2 |     for (int j = 1;j<=i ;j++ ){
3 |         cout << "*" << endl;
4 |     }
5 |     cout << endl;
6 |

```

3. 打印等腰三角形

```

1 | for (int i = 1;i<=5 ;i++ ){
2 |     for(int k=1;k<=5-i;k++){
3 |         cout << " ";
4 |     }
5 |     for (int j = 1;j<=i*2-1 ;j++ ){
6 |         cout << "*";
7 |     }
8 |     cout << endl;
9 |

```

4. 求1000以内的完数

```

1 //完数定义：若一个自然数，恰好与除去它本身以外的一切因数的和相等，这种数叫做完全数。例如，6=1+2
+3
2 for (int i=2;i<1000;i++){
3     int sum=0;
4     for (int j=1;j<i;j++){
5         if (i%j==0){
6             sum+=j;
7         }
8     }
9     if (sum==i){
10         cout << "1000之内的完整数：" << i << endl;
11     }
12 }

```

1.5. 函数

1.5.1. 函数的介绍

1.5.1.1. 函数的概念

函数，指一段可以直接被另一段程序或代码引用的程序或代码。一个较大的程序一般应分为若干个程序块，每一个模块用来实现一个特定的功能。所有的高级语言中都有子程序这个概念，用子程序实现模块的功能。

面向过程语言中，整个程序就是由函数（相互调用）组成的

面向对象语言中，函数是类的组成部份，整个程序是由很多类组成的

通俗讲，函数就是解决某件事情的办法，比如 我要上班，可以选择 步行，骑车，开车，公共交通，而每一个方式，在程序中就可能是一个函数。

1.5.1.2. 函数的组成

函数的组成要素：返回值、函数名、参数、函数体

1.5.1.3. 使用函数的好处

1. 使程序变得更简短清晰
2. 有利于程序的维护（修改）
3. 可以提高开发效率
4. 可以提高代码的重用性

1.5.2. 函数的基础使用

1.5.2.1. 函数的定义

```

1 // 函数定义的基础语法是：
2 返回值类型 函数名字(参数列表) {
3     函数体
4 }
5 // 返回值类型：表示函数执行的结果，在返回值部分详解，暂时写void
6 // 函数名字：遵循标识符的命名规则
7 // 参数列表：定义若干个参数的部分，在参数部分详解，暂时不写
8 // 函数体：函数的功能实现，在这里也业务逻辑代码

```

```

1 void sayHello() {
2     cout << "Hello!" << endl;
3 }

```

1.5.2.2. 函数的调用

函数在定义完成之后，其中的代码并不会自动的执行。如果我们需要执行函数中定义的代码，需要“调用”这个函数。

```

1 #include <iostream>
2 using namespace std;
3
4 // 定义一个函数，打印九九乘法表
5 void sayHello() {
6     for (int i = 1; i <=9 ; i++) {
7         for (int j = 1; j <=i ; j++) {
8             cout << j << "x" << i << "=" << (i*j) << "\t";
9         }
10        cout << endl;
11    }
12 }
13
14 int main() {
15     // 我们已经定义了一个函数，来打印九九乘法表，但是这个函数中的代码不会自动执行的，需要我们调用函数。
16     // 通过函数的名字来调用函数
17     sayHello();
18     return 0;
19 }
```

1.5.2.3. 函数调用的顺序

函数中，除了能进行我们的业务逻辑实现之外，还可以再调用其他的函数。而在新调用的函数中，又可以调用另外的函数。那么，函数在多次调用的时候，应该按照什么样的顺序去执行呢？我们以下方代码为例：

```

1 void print1() {
2     cout << 1 << endl;
3     print2();
4 }
5
6 void print2() {
7     print(3);
8     cout << 2 << endl;
9 }
10
11 void print3() {
12     cout << 3 << endl;
13 }
14
15 int main() {
16     cout << "start..." << endl;
17 }
```

```

18     print1();
19
20     cout << "end" << endl;
21 }
```

函数的执行，需要压到栈中去执行。

这种结构的特点是：“先进先出”

也就是先调用的函数，在栈的底部存放。而新调用的函数，会在栈的顶部存放。程序先处理栈顶的函数中的逻辑，因此最新调用的函数中的内容会执行。

```

1 上述代码的执行结果是:
2 start...
3 1
4 3
5 2
6 end
```

1.5.2.4. 参数的介绍

我们已经知道了函数是什么以及应该如何定义，也已经把部分逻辑代码段封装进了一个函数中。我们什么时候需要使用这个逻辑代码段，直接调用函数即可。

但是，有时候我们在调用函数，执行这段逻辑代码的时候，是需要有一些数据传入的。调用方必须传入指定的数据，才能够完成对应的功能。那么在这种情况下，我们应该怎么做？

```

1 例如:
2 ATM取款机，封装了取钱的功能。我们在需要取钱的时候，将银行卡插入ATM取款机，输入密码之后就可以取钱了。
3 在这个功能实现中，ATM就类似一个函数，为我们封装了一个逻辑代码段，提供了指定的功能。但是，也需要我们有
   对应的数据传入。比如，我们需要告诉这个ATM取款机，银行卡是哪个？密码是多少？我需要取多少钱等信息。
```

我们如果需要将函数之外的一些数据，带入到函数的内部，最常见的做法就是通过参数来实现。

1.5.2.5. 参数的定义

参数，其实就是一个变量。只不过，这个变量和我们之前学习的变量有不一样的地方：

1. 定义的位置不同：参数是定义在函数的参数列表小括号中的。
2. 明确数据的类型：即便是相同类型的参数，我们也需要为每一个参数明确自己的类型，不能省略类型。

```

1 void add(int num1, int num2) {
2     cout << "num1 = " << num1 << ", num2 = " << num2 << ", num1 + num2 = " << (num1
3     + num2) << endl;
```

1.5.2.6. 有参函数的调用

调用函数的时候，我们通过函数的名字添加小括号的方式来调用。但是，如果这个函数是有参数的函数，那么在调用的时候必须要明确参数的值是什么。这就类似于每一个变量必须要有值才能够使用。

调用函数的时候，必须为每一个参数进行赋值

- 将需要给参数赋的值，直接写在调用时候的小括号内即可
- 小括号内的值的数量，需要与定义时候的数量相同
- 小括号内的值的数据类型，需要与定义时候的数据类型一致
- 小括号内的值，会依次的赋值给每一个参数

```

1 // 定义了一个带有参数的函数
2 void cal(const int num1, int num2) {
3     cout << "num1 = " << num1 << ", num2 = " << num2 << ", num1 + num2 = " <<
4     (num1 + num2) << endl;
5 }
6
7 int main() {
8     // 调用函数的时候，就需要带上参数
9     cal(100, 200);
10    cal(1000, 2000);
11
12    return 0;
}

```

1.5.2.7. 几个关键名词

- **形参**：在定义函数的时候，小括号中定义的参数。由于这样的参数只有形式上的定义，并没有具体的值，因此被称为形式参数。
- **实参**：在调用函数的时候，小括号中定义的参数。由于这样的参数，为形参提供了确切的值。因此将这样的参数叫做实际参数。
- **传参**：在调用函数的时候，用实参给形参赋值，这样的过程叫做传参。

1.5.2.8. 返回值的介绍

在我们之前定义的函数中，可以通过参数将函数外的数据传递到函数的内部，那么有没有发现函数变得灵活点了吧？但是现在还有一个问题：我们如何将函数内的数据传递到函数的外部呢？

函数，其实就是对一段逻辑功能的封装。例如：我封装了一个函数，用来计算两个数字的差值绝对值。那么需要计算的两个数字，我可以使用参数传递到函数的内部，可是计算出来的差值绝对值，我应该怎么样传递出去呢？这就需要借助返回值来实现了。

1 前面在讲参数的时候，提到了一个ATM取款机的例子。我们将银行卡作为参数传递到了ATM取款机里面，在我们进行了取款的操作之后，ATM也需要将我们需要的结果（取到的钱）返回给我们。这就是将数据从函数内传递到函数的调用方那里，需要借助返回值来实现。

返回值，表示的是一个函数执行结束之后，得到的结果。

1.5.2.9. 有返回值的函数定义

```

1 // 设计一个函数，计算一个数字的绝对值
2 int abs(int number) {
3     return number > 0 ? number : -number;
4 }
```

1.5.2.10. return关键字

在函数中，return是一个非常重要的关键字！有两种作用：

1. 后面跟上某一个值，作为一个函数的执行结果，也就是返回值。
2. 结束一个函数的执行。

注意事项：

1. 在返回值类型不是void的函数中，函数执行结束之前，必须要使用return明确的返回一个结果。
2. 在返回值类型是void的函数中，仍然可以使用return关键字，此时仅表示结束函数的执行。

1.5.3. 函数的高阶使用

1.5.3.1. 函数的参数默认值

在定义参数的时候，我们可以给参数一个默认的值，例如：

```

1 int add(int num1, int num2 = 100) {
2     return num1 + num2;
3 }
```

此时在调用函数的时候，因为num2是有默认值的，因此可以给num2设置实参，也可以不设置

```

1 int main() {
2     // 给num2实参，此时的num2的值就是实参给定的值
3     cout << add(100, 200) << endl;
4     // 不给num2实参，此时的num2的值就是默认值
5     cout << add(100) << endl;
6 }
```

但是需要注意，有默认值的参数，必须放到参数列表的末尾。

```

1 // 这个是正确的
2 void cal(int num1 = 10, int num2 = 20) {}
3
4 // 这个是正确的
5 void cal(int num1, int num2 = 10) {}
6
7 // 这个是错误的！！！
8 void cal(int num1 = 10, int num2) {}

```

1.5.3.2. 函数的重载

如果在一个类中的多个函数，满足如下两点，则这两个函数就构成了重载关系(Overload)

1. 函数名相同
2. 参数不同

参数的不同，体现在数量不同、类型不同上。

注意：重载只与函数的名字、参数有关系，与返回值没有关系！

```

1 // 重载函数的定义
2 int add(int num1, int num2) {
3     return num1 + num2;
4 }
5 double add(int num1, double num2) {
6     return num1 + num2;
7 }
8 double add(double num1, int num2) {
9     return num1 + num2;
10 }
11 double add(double num1, double num2) {
12     return num1 + num2;
13 }
14
15 int main() {
16
17     // 区分调用不同的重载方法，应该从实参入手
18     cout << add(10, 20) << endl;           // 调用的是add(int, int)的方法
19     cout << add(10, 11.3) << endl;         // 调用的是add(int, double)的方法
20     cout << add(11.2, 10) << endl;         // 调用的是add(double, int)的方法
21     cout << add(11.1, 22.2) << endl;         // 调用的是add(double, double)的方法
22
23     return 0;
24 }

```

有了函数的重载，可以使得我们的程序更加的灵活！

1.5.3.3. 函数的递归

递归，是一种程序设计的思想。在解决问题的时候，可以将问题拆分成若干个小问题。这些小问题的解决方式，与大的问题解决方式相同。通过解决这些小的问题，逐渐解决这个大的问题。那么什么情况下可以使用递归呢？

- 当需要解决的问题可以拆分成若干个小问题，大小问题的解决方法相同。
- 有固定规律，方法中调用自己。

那我们应该如何正确的使用递归呢？

递归，由于涉及到方法的循环调用，因此容易出现死递归的情况。即所有的方法调用没有出口，只能将方法压栈执行，但是无法结束方法。因此，在使用递归的时候，需要设置有效的出口条件，避免无穷递归。

- 计算5的阶乘。

回归：基于出口的结果，逐层向上回归，一次计算每一层的结果，直至回归到最顶层。

递进：每一次推进，计算都比上一次变得简单，直至简单到无需继续推进，就能获得结果。也叫到达出口。

```

1 int multiply(int num) {
2     if (num == 1) {
3         return 1;
4     }
5     return num * multiply(num - 1);
6 }
```

1.5.3.4. 调用其他文件中的函数

我们程序中默认只能调用当前文件中定义的函数，但是一个大型的项目不止由一个文件来组成，很多时候是需要跨文件来调用、访问的。例如，我们将常用的数学计算的功能定义在某一个文件中，那么如何调用这个文件中定义的函数呢？

首先我们需要知道，.cpp文件中的内容是无法跨文件直接访问的，如果我们需要让某一个函数跨文件访问，需要为其定义一个.h文件，我们称为“头文件”。在头文件中添加函数的声明部分即可。需要使用的时候，直接使用#include来包含指定的头文件即可完成。

```

1 // 文件名: qfmath.h
2 int abs(int num);
3 int add(int num1, int num2);
```

```

1 // 文件名: qfmath.cpp
2 int abs(int num) {
3     return num >= 0 ? num : -num;
4 }
5
6 int add(int num1, int num2) {
7     return num1 + num2;
8 }
9
10 void test() {
11     cout << "this is a test method" << endl;
12 }
```

```

1 // 文件名: other.cpp
2 #include <iostream>
3 #include "qfmath.h"      // 包含用户自己定义的头文件，只能使用双引号，不能使用尖括号
4
5 int main() {
6     // 此时可以直接使用qfmath.h中定义的方法，实现部分写在了qfmath.cpp文件中
7     cout << abs(-10) << endl;
8     cout << add(10, 20) << endl;
9     // 由于qfmath.h中并没有定义test方法，因此无法在这里调用test方法
10    return 0;
11 }
```

1.5.4. 函数的练习

1. 设计方法，计算两个日期之间相差多少天。两个日期的年月日由参数控制。

```

1 // 设计方法，计算两个日期之间相差多少天。两个日期的年月日由参数控制。
2 int getDelta(int fromYear, int fromMonth, int fromDay, int toYear, int
3 toMonth, int toDay) {
4     // 1、起始那一天，是fromYear的第几天
5     // 2、终止那一天，是toYear的第几天
6     // 3、计算 fromYear的1月1日 ~ toYear的1月1日相差多少天
7     // 4、3 + 2 - 1
8
9     int fromDays = getDays(fromYear, fromMonth, fromDay);
10    int toDays = getDays(toYear, toMonth, toDay);
11
12    int days = 0;
13    for (int y = fromYear; y < toYear; y++) {
14        days += check(y) ? 366 : 365;
15    }
16
17    return days + toDays - fromDays;
18 }
```

```

18 // 计算一个日期是当年的第几天
19 int getDays(int year, int month, int day) {
20     int days = day;
21
22     for (int m = 1; m < month; m++) {
23         if (m == 1 || m == 3 || m == 5 || m == 7 || m == 8 || m == 10 || m ==
24 12) {
25             days += 31;
26         }
27         else if (m == 4 || m == 6 || m == 9 || m == 11) {
28             days += 30;
29         }
30         else if (m == 2) {
31             days += check(year) ? 29 : 28;
32         }
33     }
34
35     return days;
36 }
37
38 // 验证一个年份是不是一个闰年
39 static boolean check(int year) {
40     return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
41 }
```

2. 开发一个标题为“FlipFlop”的游戏应用程序。它从 1 计数到 100，遇到 3 的倍数就替换为单词“Flip”，5 的倍数就替换为单词“Flop”，既为 3 的倍数又为 5 的倍数则替换为单词“FlipFlop”

```

1 void flipFlop() {
2     for (int i = 1; i <= 100; i++) {
3         if (i % 3 == 0 && i % 5 == 0) {
4             cout << "FlipFlop" << endl;
5         }
6         else if (i % 3 == 0) {
7             cout << "Flip" << endl;
8         }
9         else if (i % 5 == 0) {
10            cout << "Flop" << endl;
11        }
12        else {
13            cout << i << endl;
14        }
15    }
16 }
```

3. 两个自然数X，Y相除，商3余10，被除数、除数、商、余数的和是163，求被除数、除数。

```

1 void test() {
2     for (int x = 0; x < 163; x++) {
3         for (int y = 1; y < 163; y++) {
4             // 求商
5             int s = x / y;
6             // 求余
7             int l = x % y;
8
9             if (s == 3 && l == 10 && (x + y + s + l == 163)) {
10                 cout << "x = " << x << ", y = " << y << endl;
11             }
12         }
13     }
14 }
```

4. 使用递归计算 $1+2+3+\dots+100$ 的和

```

1 int sum(int num) {
2     if (num == 1) {
3         return 1;
4     }
5     return num + sum(num - 1);
6 }
```

5. 使用递归输出30位的斐波那契数列

```

1 int getFibonacci(int index) {
2     if (index == 1 || index == 2) {
3         return 1;
4     }
5     return getFibonacci(index - 1) + getFibonacci(index - 2);
6 }
```

1.6. 指针与引用

1.6.1. 内存分析

1.6.1.1. 内存分区

程序在执行的时候，会在内存中开辟一些空间，存储一些数据。而内存又可以分为四个分区：

- 栈区
- 堆区
- 全局区
- 代码区

1.6.1.2. 各分区作用

- 代码区

1 | 存放程序编译之后生成的二进制代码，例如我们写的函数，就是存储在这里的。
 2 | PS：函数在程序编译之后，存储于代码区。调用函数的时候，会压到栈区执行其中的代码。

- 全局区

1 | 全局区内的变量在程序编译阶段已经分配好内存空间并初始化。这块内存在程序的整个运行期间都存在，主要存放静态变量、全局变量和常量。

```

1 #include <iostream>
2
3 using namespace std;
4
5 // 定义全局变量
6 int g_num1 = 100;
7 // 定义全局常量
8 const int g_num2 = 100;
9 // 定义全局静态变量
10 static int g_num3 = 100;
11 // 定义全局静态常量
12 static const int g_num4 = 100;
13
14 int main() {
15
16     // 定义局部变量
17     int num1 = 10;
18     // 定义局部常量
19     const int num2 = 10;
20     // 定义局部静态变量
21     static int num3 = 10;
22     // 定义局部静态常量
23     static const int num4 = 10;
24
25     cout << &g_num1 << endl;          // 全局区
26     cout << &g_num2 << endl;          // 全局区
27     cout << &g_num3 << endl;          // 全局区
28     cout << &g_num4 << endl;          // 全局区
29
30     cout << &num1 << endl;           // 栈区
31     cout << &num2 << endl;           // 栈区
32     cout << &num3 << endl;           // 全局区
33     cout << &num4 << endl;           // 全局区
34
35     return 0;
36 }
```

- 栈区

1 | 由系统进行内存的管理。主要存放函数的参数以及局部变量。在函数完成执行，系统自行释放栈区内存，不需要用户管理。

```
1 #include <iostream>
2
3 using namespace std;
4
5
6 int* getNumber() {
7     int number = 100;
8     cout << "&number = " << &number << endl;
9     return &number;
10 }
11
12 int main() {
13
14     int* p = getNumber();
15     cout << "p = " << p << endl;
16     cout << "*p = " << *p << endl;
17
18     return EXIT_SUCCESS;
19 }
```

- 堆区

1 | 由编程人员手动申请，手动释放，若不手动释放，程序结束后由系统回收，生命周期是整个程序运行期间。

```
1 #include <iostream>
2
3 using namespace std;
4
5
6 int* getNumber() {
7     int* number = new int(100);
8     cout << "number = " << number << endl;
9     return number;
10 }
11
12 int main() {
13
14     int* p = getNumber();
15     cout << "p = " << p << endl;
16     cout << "*p = " << *p << endl;
```

```

17
18     return EXIT_SUCCESS;
19 }
```

1.6.1.3. 内存中的数据残留

- 1 我们在硬盘上存储了很多的数据，假如你不小心误删除了一个数据文件，想要恢复。你请教了身边的大神，怎么恢复这些数据啊？大神跟你说，你可以借助一些数据恢复的工具进行操作，但是从现在开始，不要再使用这个硬盘了！
- 2 为什么大神不让你再用这个硬盘了呢？？
- 3 原因在于，其实我们所谓的“删除数据”，删除的只是你对指定地址范围空间的“使用权”，你不能够再去使用这块空间了，但是这块空间中原来的内容是不会被删除掉的！所谓的数据恢复的软件就是帮你找到之前你使用的这个空间的地址范围是什么，然后读取这里的内容即可。但是为什么误删除掉数据之后，你就不能再使用这个硬盘了呢？
- 4 如果你再次使用这个硬盘去存储文件，新的文件需要开辟一块空间来存储吧，那么就有可能存储在你之前释放的那一部分，而此时新的数据文件会将你原来的数据文件给覆盖掉，这个时候，你再想找回原来的数据，难度就非常大了！

通过上述案例我们知道，其实所谓的“删除”，只是释放掉了你对某一块空间的使用权。在硬盘上是这样的，在内存中也是如此。我们定义了一个变量，此时会在内存中开辟了一块空间出来。那么这个空间中还残留着之前的数据呢！

```

1 int main() {
2     int a;
3     cout << a << endl; // 是不是发现一些很奇怪的值?
4     // 局部变量a是在栈空间中开辟，栈空间内存不需要我们手动管理，开辟新的空间的时候，也不会进行初始化
5     // 的操作。
6
7     int* p = new int();
8     cout << *p << endl; // 此时可以看到是0
9     // new表示在堆空间中开辟内存，同时会进行初始化操作，将这块内存中原来的值给覆盖掉，初始化为0。
}
```

1.6.2. 指针的介绍

我们在前面的代码中定义了一些变量，其实定义变量就是在内存中开辟了一块指定大小的空间，空间开辟的大小取决于不同的数据类型所占用的空间大小。并且可以在这样的空间中进行值的填充。那么指针是什么呢？每一个开辟中的内存空间，都是有一个唯一的地址的，而这样的地址我们就称为是“指针”。

```

1 如果我们把计算机的内存比作是一个酒店，在这个酒店中有很多的房间，这些房间有不同的类型，有标准间、大床房、套房、亲子房等等，每一个房间也都起了自己的名字，例如听云轩、风雅阁等，每一个房间也都有自己的门牌号，例如1001、1202等，每一个房间中也都住有不同的客人。
2 在这里我们做一个类比：
3 酒店      计算机内存
4 房间      内存中开辟的一块空间
5 房间的名字  变量的名字
6 房间门牌号  内存的地址（指针）
7 房间的住客  空间中存储的值，也就是这个变量的值
8
9 例如：int number = 10;
10 这段代码执行的时候，会在内存中开辟出4字节的空间，命名为number，这个空间中存储的值为10
11 而指针是什么呢？指针就是number的内存地址，也就是这个酒店房间的门牌号

```

1.6.3. 指针变量的定义

```

1 // 定义了一个int类型的变量num，值为100
2 // 也就是在内存中开辟了一个4字节的空间，存储的值为100
3 int num = 100;
4
5 // 那么，num的地址怎么获取呢？
6 // &num 就可以获取到变量num的地址！
7 // 那么，这个地址应该用什么类型的变量来接收呢？
8 int* p = &num;
9
10 // 我们可以直接输出指针类型的变量p中存储的内存地址
11 cout << p << endl; // 输出的就是一个内存地址，一般是用一个十六进制的数字来表示
12
13 // 那么我们如何通过这个地址来找到它指向的空间呢？
14 // *p
15 cout << *p << endl; // 输出的就是指针变量p中存储的地址所指向的空间中存储的值！也就是变量num的值！

```

1.6.4. 空指针与野指针

1.6.4.1. 空指针

空指针，指的是没有存储任何内存地址的指针变量，一般使用NULL来表示一个空的地址。通常情况下，使用空指针可以对指针变量进行初始化。

```

1 int* p = NULL; // 这里的指针变量p没有存储任何的地址，就是一个空指针。
2
3 // 设置为NULL的指针变量，存储的地址其实是0，但是地址0到255的内存为系统内存，不允许访问
4 // 因此这里会出现“读取访问权限”的问题
5 cout << *p << endl;

```

1.6.4.2. 野指针

野指针中存储有一个内存地址，但是这个地址指向的空间已经不存在了。在这种情况下，这个指针变量中存储的地址已经没有什么意义了，这样的指针称为野指针。

```

1 // 这里定义了一个指针变量，随便写了一个地址，这个地址对应的空间极有可能是不存在的
2 int* p = (int*)0x1234;
3 // 访问野指针，也是会出现问题的
4 cout << *p << endl;
```

1.6.5. 常量指针与指针常量

- 常量指针
 - const放在*之前，表示常量指针，即常量的指针
 - 指针的指向是可以修改的，但是不能通过指针来修改指向空间的值
- 指针常量
 - const放在*之后，表示指针常量，即指针是一个常量值
 - 可以通过指针来修改指向空间的值，但是不能修改指针的地址指向

```

1 int num1 = 100;
2 int num2 = 200;
3
4 // const放在*之前，表示常量指针，即常量的指针
5 // 指针的指向是可以修改的，但是不能通过指针来修改指向空间的值
6 const int* p1 = &num1;
7 p1 = &num2;
8 // *p1 = 200;
9
10 // const放在*之后，表示指针常量，即这个指针是一个常量
11 // 可以通过指针修改指向空间的值，但是指针的指向是不可以修改的
12 int* const p2 = &num1;
13 // p2 = &num2;
14 *p2 = 300;
```

1.6.6. 指针在函数中的使用

```

1 #include <iostream>
2
3 using namespace std;
4
5 void changeNumber(int number) {
6     number = 100;
7 }
```

```

8
9 void changeNumber(int* number) {
10     *number = 100;
11 }
12
13 int main() {
14     int num = 10;
15
16     // 这里是值传递，传递到函数changeNumber中的是10，因此在函数中修改参数number的值，对这里的变量num没有影响
17     changeNumber(num);
18     cout << "num = " << num << endl;
19
20     // 这里是地址传递，传递到函数changeNumber中的是变量num的地址，在函数中通过地址修改指向空间的
21     // 值，这里的num会受到影响
22     changeNumber(&num);
23     cout << "num = " << num << endl;
24
25     return 0;
26 }
```

1.6.7. 引用的介绍

变量名实质上是一段连续内存空间的别名，是一个标号(门牌号)，程序中通过变量来申请并命名内存空间，通过变量的名字可以使用存储空间。

对一段连续的内存空间只能取一个别名吗？

c++中新增了引用的概念，引用可以作为一个已定义变量的别名。

基本语法：

```
Type& ref = val;
```

注意事项：

- &在此不是求地址运算，而是起标识作用。
- 类型标识符是指目标变量的类型
- 必须在声明引用变量时进行初始化。
- 引用初始化之后不能改变。
- 不能有NULL引用。必须确保引用是和一块合法的存储单元关联。
- 可以建立对数组的引用。

1.6.8. 引用的基础使用

```

1 int main() {
2     // 定义一个整型变量
3     int num = 10;
4     // 定义一个num的引用（起一个别名）
```

```

5 int& a = num;
6
7 // 比较值
8 cout << "num = " << num << ", a = " << a << endl;
9
10 // 比较地址, 可以看到地址是完全相同的, 也就是指向的同一块空间
11 cout << "&num = " << &num << ", &a = " << &a << endl;
12
13 // 因为指向的是同一块空间, 因此对任意一个的修改, 都会影响到另外的一个
14 num = 200;
15 cout << "num = " << num << ", a = " << a << endl;
16 }

```

1.6.9. 引用在函数中的使用

```

1 void changeNumber1(int n) {
2     n = 200;
3 }
4
5 void changeNumber2(int& n) {
6     n = 200;
7 }
8
9 int main() {
10     int number = 10;
11
12     changeNumber1(number);
13     cout << "number = " << number << endl; // 值传递, 因此这里的number并没有发生变化
14
15     changeNumber2(number);
16     cout << "number = " << number << endl; // 引用传递, 因此这里的number也会发生变化
17
18     return 0;
19 }

```

1.6.10. 引用的本质

所谓的引用, 其其实质来讲就是一个指针常量。

```

1 int main() {
2     // 定义一个整型的变量n
3     int n = 10;
4
5     // 定义n的引用
6     // 这里相当于是 int* const rn = &a;
7     // 这也就解释通了为什么rn和a引用的同一块空间，并且为什么rn不能修改引用。
8     int& rn = n;
9
10    // 在通过引用进行空间访问的时候，系统会自动的转换成 *rn = 200;
11    rn = 200;
12 }

```

1.6.11. 常量引用

常量引用，就是对一个常量建立引用，又称为“常引用”。主要用在函数的形参部分，访问误操作导致在函数题中通过形参，修改实参的值

```

1 void change(const int& n) {
2     // 这里会出问题
3     // n = 200;
4     cout << n << endl;
5 }

```

1.7. 数组

1.7.1. 数组的介绍

1.7.1.1. 数组是什么

数组其实就是一个数据容器，里面可以存储若干个相同的数据类型的数据。

```

1 小案例：
2 现在有100个学生的成绩需要存储，那么怎么存储呢？
3 以我们现有的知识储备来说，可以定义100个变量来存储这些数据，但是这样是非常麻烦的，重复的操作特别多。
4 而且，如果我需要对这些成绩进行批量的操作呢？例如，我需要将这些成绩都加1分？
5 如果我们使用100个变量来存储的话，那么需要依次修改每一个变量的值，这是一个很大的工作量。。
6 那么有没有一个比较简单的方式呢？
7
8 数组！
9 我们可以把这100个成绩存入一个数组中存储起来！此时只需要定义一个数组类型的变量即可，省去了100个变量定义这样的重复的操作。
10 而且，如果需要对数组中的这100个成绩进行批量的操作，直接用循环遍历数组，依次对数据进行修改即可。

```

1.7.1.2. 数组的特性

- 数组可以用来存储任意数据类型的数据，但是所有的数据需要是相同的数据类型。
- 数组是一个定长的容器，一旦初始化完成，长度将不能改变。

1.7.1.3. 数组中的几个名词

元素：数组中存储的每一个数据，称为数组中的元素。

长度：数组的容量，即数组中可以存储多少个元素。

遍历：依次获取数组中的每一个元素。

1.7.2. 数组的基础使用

1.7.2.1. 数组的定义

```

1 // 1. 定义指定长度的数组，此时数组中填充的元素是不安全的。
2 int array1[10];
3
4 // 2. 定义指定长度的数组，并使用默认的值来填充
5 int array2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};           // 定义一个存储int类型元素的数
6 组，填充10个初始的值。
7 int array3[10] = {1, 2, 3};      // 如果大括号中的初始的元素数量小于数组长度，剩余的元素填充默
8 认值。
9 // 3. 定义一个数组，指定数组中的元素，此时数组的长度由初始元素的数量来决定
10 int array4[] = {1, 2, 3, 4, 5};

```

1.7.2.2. 数组的访问

为了能够区分数组中存储的每一个元素，在数组中存储的每一个元素都有一个唯一的序号，称为 **下标**。我们在访问数组中的元素的时候，通过下标来访问。

注意事项：数组中元素的下标是从**0**开始的！即数组中的元素下标范围是 **【0, 数组长度-1】**

```

1 // 定义一个数组
2 int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3
4 // 访问下标为5的元素
5 cout << array[5] << endl;          // 读取值并输出
6 array[5] = 50;                     // 修改指定下标的元素的值
7
8 // 数组长度的获取
9 int length = sizeof(array) / sizeof(int);
10
11 // 遍历数组，并输出遍历到的元素

```

```

12 for (int i = 0; i < length; i++) {
13     cout << array[i] << endl;
14 }
```

注意事项：通过下标访问数组元素的时候，注意不要越界！

1.7.3. 数组的内存分析

数组是一个容器，在内存中进行空间开辟的时候，并不是一个整体的空间，而是开辟了若干个连续的空间。

例如：int array[10];

这个数组的长度为10，存储的元素数据类型是int。也就是说，需要在内存中开辟连续的10个4字节空间来存储元素。

而array表示什么呢？表示的是数组中首元素的内存地址！

```

1 int arr[10] = {1, 2, 3, 4, 5};
2 在上述代码中，我们定义了一个长度为10的数组。其中arr表示的是数组中首元素的内存地址。
3 于是，我就可以直接通过 *arr 来找到数组中的首元素，那么后面一个元素呢？
4 我们可以通过arr的内存地址+4来访问到，再后面的一个元素，再加一个4
5 （为什么要加4呢？因为这个数组中存储的元素类型是int，占据4个字节空间。如果是一个short数组，那就需要
+2了）
6
7 但是每次都需要我们手动的计算内存地址有点麻烦，因此C++将这一个元素的内存占用空间大小作为一个“单位”
8 （例如：int数组，一个单位就是4个字节，short数组，一个单位就是2个字节）
9
10 在进行元素访问的时候，
11 首元素直接通过arr就可以访问，
12 后面的一位元素，偏移一个单位的地址！arr + 1个单位
13 再后面的一位元素，偏移两个单位的地址！arr + 2个单位
14 再后面的一位元素，偏移三个单位的地址！arr + 3个单位
15
16 这就是为什么，数组中元素的下标是从0开始的！
```

注意事项：当数组作为参数传递到一个函数中的时候，传递的只是首元素的地址！

```

1 // 这里的参数其实等价于 int* arr, 只是一个指向首元素的地址
2 void printArray1(int arr[]) {
3     // 计算数组的长度
4     int len = sizeof(arr) / sizeof(int);
5     cout << len << endl;      // 2
6 }
7
8 // 因此需要在一个函数中完成数组的遍历、排序等操作的时候，需要带上长度
9 void printArray2(int* arr, int len) {
10    for (int i = 0; i < len; i++) {
11        cout << arr[i] << endl;
12    }
13 }
```

1.7.4. 数组的遍历

1.7.4.1. 下标遍历

```

1 int arr[10] = {1, 2, 3, 4, 5, 6};
2
3 // 计算长度
4 int len = sizeof(arr) / sizeof(int);
5
6 // 遍历
7 for (int i = 0; i < len; i++) {
8     cout << arr[i] << endl;
9 }
```

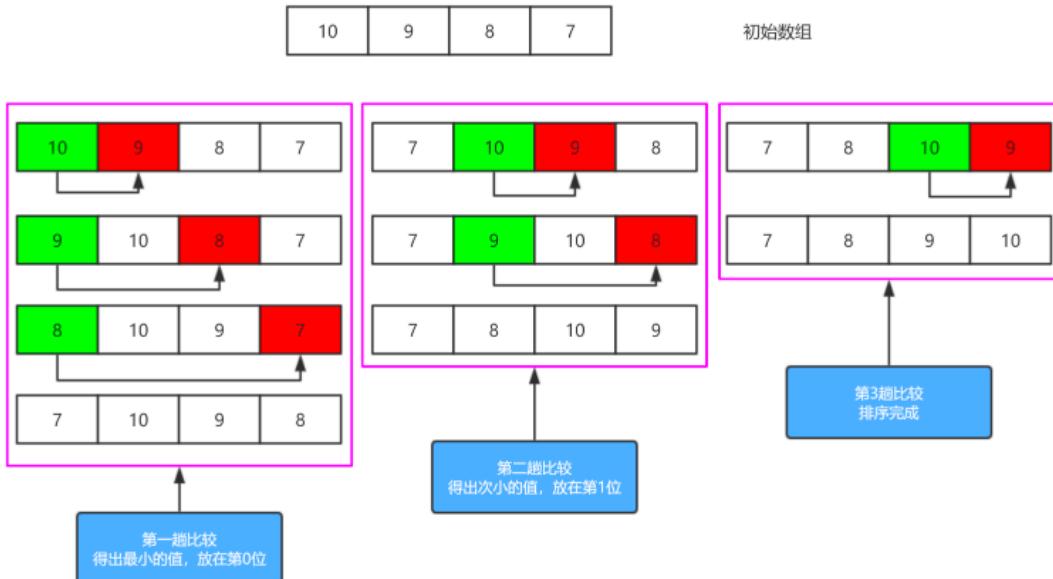
1.7.4.2. 范围遍历

```

1 int arr[10] = {1, 2, 3, 4, 5};
2
3 for (int ele : arr) {
4     cout << ele << endl;
5 }
```

1.7.5. 数组的排序

1.7.5.1. 选择排序



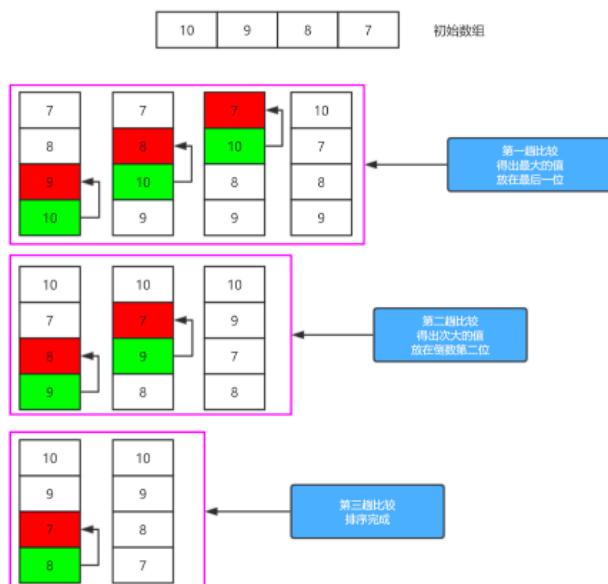
```

1 #include <iostream>
2
3 using namespace std;
4
5 void sort01(int* array, int len) {
6     // 固定下标
7     for (int i = 0; i < len - 1; i++) {
8         // 定义最小值下标
9         int minIndex = i;
10        // 遍历i之后的每一个元素
11        for (int j = i + 1; j < len; j++) {
12            if (array[minIndex] > array[j]) {
13                // 找到了新的最小值，更新最小值下标
14                minIndex = j;
15            }
16        }
17        // 内部循环结束后，交换当前遍历的元素和记录的最小值下标位的元素
18        if (minIndex != i) {
19            int tmp = array[minIndex];
20            array[minIndex] = array[i];
21            array[i] = tmp;
22        }
23    }
24 }
25
26 void printArray(int* array, int len) {
27     for (int i = 0; i < len; i++) {
28         cout << array[i] << "\t";
29     }
30 }
31
32 int main() {
33     // 定义一个数组
34 }
```

```

35     int array[10] = {1, 3, 5, 7, 9, 0, 8, 6, 4, 2};
36     // 计算数组长度
37     int len = sizeof(array) / sizeof(int);
38
39     // 选择排序
40     sort01(array, len);
41
42     // 输出排序之后的数组
43     printArray(array, len);
44
45
46     return EXIT_SUCCESS;
47 }
```

1.7.5.2. 冒泡排序



```

1 #include <iostream>
2
3 using namespace std;
4
5 void sort01(int* array, int len) {
6     for (int i = 0; i < len - 1; i++) {
7         for (int j = 0; j < len - 1 - i; j++) {
8             if (array[j] > array[j + 1]) {
9                 int tmp = array[j];
10                array[j] = array[j + 1];
11                array[j + 1] = tmp;
12            }
13        }
14    }
15 }
```

```

17 void printArray(int* array, int len) {
18     for (int i = 0; i < len; i++) {
19         cout << array[i] << "\t";
20     }
21 }
22
23 int main() {
24
25     // 定义一个数组
26     int array[10] = {1, 3, 5, 7, 9, 0, 8, 6, 4, 2};
27     // 计算数组长度
28     int len = sizeof(array) / sizeof(int);
29
30     // 选择排序
31     sort01(array, len);
32
33     // 输出排序之后的数组
34     printArray(array, len);
35
36
37     return EXIT_SUCCESS;
38 }
```

1.7.6. 数组元素查找

数组元素查找指的是从给定的一个数组中查询指定元素出现的下标。

由于需要查询的元素在指定的数组中可能出现多次，在这里我们只需要找到一个即可。

1.7.6.1. 顺序查询法

1 | 顺序查询，就是从前往后遍历数组，将数组中的每一个元素和需要查询的元素进行对比。如果比较结果是相同的，说明找到了需要查询的元素。

```

1 #include <iostream>
2
3 using namespace std;
4
5 /**
6  * 从一个数组中查询指定的元素element出现的下标
7  * @param array 需要查询元素的数组
8  * @param len 数组的长度
9  * @param element 需要查找的元素
10 * @return 元素存在的下标，如果不存在这个元素，返回-1
11 */
12 int indexOf(const int* array, int len, int element) {
13     // 遍历数组中的每一个元素
14     for (int i = 0; i < len; i++) {
```

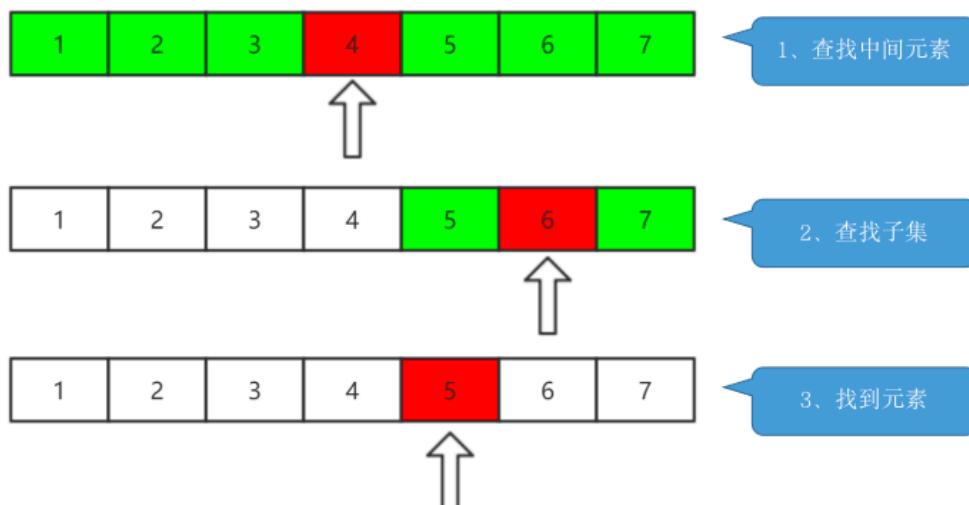
```

15     if (array[i] == element) {
16         return i;
17     }
18 }
19 return -1;
20 }
21
22 int main() {
23
24     // 定义一个数组
25     int array[10] = {1, 3, 5, 7, 9, 0, 8, 6, 4, 2};
26     // 计算数组长度
27     int len = sizeof(array) / sizeof(int);
28
29     // 从数组中查询元素
30     int index = indexOf(array, len, 7);
31     cout << index << endl;
32
33     return EXIT_SUCCESS;
34 }
```

1.7.6.2. 二分查询法

二分查询，即利用数组中间的位置，将数组分为前后两个子表。如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

二分查询，要求数组必须是排序的，否则无法使用二分查询。



```

1 #include <iostream>
2
3 using namespace std;
4
```

```
5  /**
6   * 从一个数组中查询指定的元素element出现的下标
7   * @param array 需要查询元素的数组, 升序排序的数组
8   * @param len 数组的长度
9   * @param element 需要查找的元素
10  * @return 元素存在的下标, 如果不存在这个元素, 返回-1
11 */
12 int indexOf(const int* array, int len, int element) {
13
14     // 定义变量, 确定需要查询的范围
15     int minIndex = 0, maxIndex = len - 1;
16     while (minIndex <= maxIndex) {
17         // 计算出新的中间下标
18         int midIndex = (maxIndex + minIndex) / 2;
19         // 比较中间值和需要查询的元素
20         if (array[midIndex] > element) {
21             // 中间值比要查询的元素大, 说明元素在中间值的左边
22             maxIndex = midIndex - 1;
23         } else if (array[midIndex] < element) {
24             // 中间值比要查询的元素小, 说明元素在中间值的右边
25             minIndex = midIndex + 1;
26         } else {
27             // 说明找到这个元素了
28             return midIndex;
29         }
30     }
31
32     return -1;
33 }
34
35 int main() {
36
37     // 定义一个数组
38     int array[10] = {1, 3, 5, 7, 9, 0, 8, 6, 4, 2};
39     // 计算数组长度
40     int len = sizeof(array) / sizeof(int);
41
42     // 从数组中查询元素
43     int index = indexOf(array, len, 7);
44     cout << index << endl;
45
46     return EXIT_SUCCESS;
47 }
```

1.7.7. 数组的练习

1. 设计一个函数，找出一个数组中最大的数字，连同所在的下标一起输出。

```

1 void printMaxElementAndIndex(int* array, int len) {
2     // 空数组判断
3     if (len == 0) {
4         cout << "空数组，不存在最大值! " << endl;
5         return;
6     }
7     // 不是空数组，假设数组中第0个元素就是最大的
8     int maxElement = array[0], maxIndex = 0;
9     // 遍历数组，依次和当前记录的最大值进行比较
10    for (int i = 0; i < len; i++) {
11        if (array[i] > maxElement) {
12            // 遍历到了新的最大值
13            maxElement = array[i];
14            maxIndex = i;
15        }
16    }
17    // 输出结果
18    cout << "最大值是: " << maxElement << ", 所在的下标是: " << maxIndex << endl;
19 }
```

2. 设计一个函数，判断一个数组是不是一个升序的数组。

```

1 bool checkAscending(const int* array, int len) {
2     // 思路：从前往后，依次比较两个相邻的元素，如果后面的元素比前面的小，就可以说明不是升序
3     for (int i = 0; i < len - 1; i++) {
4         if (array[i] > array[i + 1]) {
5             return false;
6         }
7     }
8     return true;
9 }
```

```

1 bool checkAscending(const int* array, int maxIndex) {
2     // 递归：
3     if (maxIndex == 1) {
4         return array[1] >= array[0];
5     }
6
7     // 如果maxIndex位的元素大于等于maxIndex-1位的元素，并且前maxIndex-1位的元素是升序的，那么整体就是升序的
8     return array[maxIndex] >= array[maxIndex - 1] && checkAscending(array,
9     maxIndex - 1);
}
```

3. 设计一个函数，找出一个整型数组中的第二大的值。

1. 不可以通过排序实现，不能修改数组中的数据顺序
2. 要考虑到最大的数字可能出现多次

```

1 int getSecondMax(const int* array, int len) {
2     if (len <= 1) {
3         cout << "没有次大值" << endl;
4         return -1;
5     }
6     // 定义两个变量，分别用来记录最大值和次大值
7     int max = array[0], second = array[0];
8     // 遍历数组中的每一个元素
9     for (int i = 0; i < len; i++) {
10         if (array[i] > max) {
11             // 新的最大值出现了，需要更新最大值和次大值
12             second = max;
13             max = array[i];
14         } else if (array[i] < max && array[i] > second) {
15             // 新的次大值出现了
16             second = array[i];
17         }
18     }
19
20     return second;
21 }
```

4. 设计一个函数，将一个数组中的元素倒序排列（注意，不是降序）。

```

1 void reverse(int* array, int len) {
2     // 交换第0位和最后一位，第一位和倒数第二位... 交换到一半即可
3     for (int i = 0; i < len / 2; i++) {
4         int temp = array[i];
5         array[i] = array[array.length - 1 - i];
6         array[array.length - i - 1] = temp;
7     }
8 }
```

5. 将一个数组中的元素拷贝到另外一个数组中。

```

1 void copy(int* src, int srcLen, int* dst, int dstLen) {
2     // 1. 遍历原数组，依次将元素拷贝到目标数组中
3     for (int i = 0; i < srcLen; i++) {
4         // 2. dst越界判断
5         if (i >= dstLen) {
6             break;
7         }
8         dst[i] = src[i];
9     }
10 }
```

6. 设计一个函数，比较两个数组中的元素是否相同（数量、每一个元素都相同，才认为是相同的数组）。

```

1 bool equals(int* array1, int arr1Len, int* array2, int arr2Len) {
2     // 特殊判断
3     if (array1 == NULL || array2 == NULL || arr1Len != arr2Len) {
4         return false;
5     }
6     // 逐个元素进行比较
7     for (int i = 0; i < arr1Len; i++) {
8         if (array1[i] != array2[i]) {
9             return false;
10        }
11    }
12    return true;
13 }
```

1.7.8. 浅拷贝与深拷贝

有时候我们对数组进行操作的时候，需要进行数组的拷贝，而此时会有浅拷贝和深拷贝两种数组的拷贝形式。

- 浅拷贝：也就是地址拷贝，拷贝到的是数组的首元素地址。
- 深拷贝：定义一个新的数组，长度与原来的数组相同，将原来数组中的每一个元素依次拷贝到新的数组中。

从上述的说明中，可以看出，所谓的浅拷贝其实就是拷贝了一个地址，得到的数组与原来的数组指向的其实是同一块空间。因此对一个数组进行的操作都会对另外一个数组产生影响。而深拷贝则不然，深拷贝是创建了一个全新的数组，虽然元素与原来的数组元素相同，但是从内存上来看的话，这是一个全新的数组，修改这个数组不会对另外一个数组产生任何影响。

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6
7     // 定义一个需要拷贝的数组
```

```

8     int array[] = {1, 2, 3, 4, 5};
9
10    // 浅拷贝、地址拷贝
11    int* array_copy_1 = array;
12
13    // 深拷贝，创建一个新的等长的数组，并将元素依次拷贝过来
14    int array_copy_deep[5];
15    for (int i = 0; i < 5; i++) {
16        array_copy_deep[i] = array[i];
17    }
18
19    return EXIT_SUCCESS;
20 }
```

1.7.9. 二维数组

1.7.9.1. 二维数组的介绍

数组其实就是一个容器，存储着若干的数据。数组中可以存储任意类型的元素，可以存储整数、可以存储浮点数字、可以存储字符串，那么数组中能不能存储一个数组呢？？

可以的！

1 我们将数组比作是一个容器，例如我们喝水的杯子。杯子是一个容器，里面可以装水。但是杯子被厂家生产完成后，需要将若干个杯子装到一个箱子里发货吧。那么这个箱子也是一个容器，里面存放了什么呢？小容器！

如果一个数组中存储的元素类型是一个数组，那么这样的数组就是**二维数组**。

理论上讲还有三维数组、四维数组，只不过一般不去讨论。我们在讨论多维数组的时候，基本也就是指的二维数组了。

1.7.9.2. 二维数组的定义

通常我们会将二维数组比作一个行列矩阵，二维数组有多少元素，相当于有多少行。二维数组中的小一维数组有多少元素，相当于有多少列。按照这样的类比，二维数组的定义有如下几种方式：

```

1 数据类型 标识符[行数][列数];
2 数据类型 标识符[行数][列数] = { {val1, val2, val3}, {val1, val2, val3} };
3 数据类型 标识符[行数][列数] = { val1, val2, val3, val4 };
4 数据类型 标识符[][列数] = { val1, val2, val3, ... }
```

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
```

```

7 // 定义二维数组:
8 // 数据类型 标识符[行数][列数];
9 int array1[3][5];
10
11 // 数据类型 标识符[行数][列数] = { {val1, val2, val3}, {val1, val2, val3} };
12 int array2[3][5] = {
13     {1, 2, 3, 4, 5},
14     {2, 2, 3, 3, 3},
15     {3, 2, 1, 4, 5}
16 };
17
18 // 数据类型 标识符[行数][列数] = { val1, val2, val3, val4 };
19 // 此时系统会将这些元素中, 每5个元素组合到一起
20 // 最后剩余不到5个的元素, 补0凑够5位拼成一个数组
21 int array3[3][5] = {1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 200};
22
23 // 数据类型 标识符[][列数] = { val1, val2, val3, ... }
24 int array4[][][5] = {1, 2, 3, 4, 5, 2, 2, 3, 4, 5, 1};
25
26 return EXIT_SUCCESS;
27 }
```

1.7.9.3. 二维数组的使用

二维数组中的元素访问与一维数组是相同的，通过下标来进行访问即可！

第二部分 面向对象

2.1. 面向对象介绍

2.1.1. 面向对象与面向过程

- 面向过程
 - 是一种看待问题、解决问题的思维方式，着眼点在于问题是如何一步步的解决的，然后亲力亲为的解决问题。
- 面向对象
 - 是一种看待问题、解决问题的思维方式，着眼点在于找到一个能够帮助解决问题的实体，然后委托这个实体来解决问题。

2.1.2. 案例分析

2.1.2.1. 小明买电脑

- 面向过程

1	1. (小明)去市场买配件
2	2. (小明)将零件运回家里
3	3. (小明)将电脑组装起来

- 面向对象

1	1. 找到一个能够帮助买电脑的朋友 -- 老王
2	2. 委托老王去买电脑配件
3	3. 委托老王把电脑运回来
4	4. 委托老王把电脑组装起来

2.1.2.2. 把大象装冰箱

- 面向过程

1	1. (我)打开冰箱门
2	2. (我)把大象装进冰箱
3	3. (我)关上冰箱门

- 面向对象

1	1. 冰箱, 开门
2	2. 大象, 进去冰箱里
3	3. 冰箱, 关门

2.1.3. 类与对象

在面向对象的编程思想中，着眼点在于找到一个能够帮助解决问题的实体，然后委托这个实体解决问题。

在这里，这个具有特定的功能，能够解决特定问题的实体，就是一个对象。

由若干个具有相同的特征和行为的对象的组成的集合，就是一个类。

类是对象的集合，对象是类的个体。

2.2. 类的设计与对象的创建

2.2.1. 类的设计

从若干个具有相同的特征和行为的对象中，提取出这些相同的特征和行为，设计为一个类。

类中定义所有的对象共有的特征和行为，其中，特征用属性表示，行为用方法表示。

所谓属性，其实就是定义在类中的一个变量。

```
1 // 设计一个类，描述人
2 // 属性：姓名、性别、年龄
3 // 方法：走路、吃饭
4 class Person {
5 public:
6     string name;
7     string gender;
8     int age;
9
10    void walk() {
11        cout << "人类会走路" << endl;
12    }
13    void eat() {
14        cout << "人类会吃饭" << endl;
15    }
16 }
```

注意事项：

在类中定义的属性、方法，默认都是private的权限，在类外是不能访问的。如果需要在类外访问，需要修改为public权限。

//

public：在任意位置都可以访问

protected：在当前类和子类中可以访问

private：只能在当前类中访问

2.2.2. 对象的创建

```

1 int main() {
2     // 1. 直接创建对象，隐式调用
3     Person xiaobai;
4
5     // 2. 显式调用
6     Person xiaobei = Person();
7
8     // 3. 关键字new
9     Person* xiaowang = new Person();
10
11    return 0;
12 }

```

在上述的三种创建方式中，前两种方式是类似的。我们在创建对象的时候，区别主要是有没有使用关键字new。

	使用new	没有使用new
内存方面	在堆空间开辟	在栈空间开辟
内存管理	需要手动使用delete销毁	不需要手动销毁
属性初始化	自动有默认的初始值	没有初始值
语法	需要用类*来接收变量	不需要使用*
成员访问	通过.访问	通过->访问

2.2.3. 成员访问

成员访问，即访问类中的成员（属性、方法）。

```

1 int main() {
2     // 创建Person对象
3     Person xiaobai;
4
5     // 访问类中的属性
6     xiaobai.name = "xiao bai";
7     xiaobai.age = 10;
8
9     // 访问类中的方法
10    xiaobai.walk();
11    xiaobai.eat();
12 }

```

```

1 int main() {
2     // 使用new创建对象
3     Person* xiaobai = new Person();
4
5     // 访问类中的属性
6     xiaobai -> name = "xiao bai";
7     xiaobai -> age = 1;
8
9     // 访问类中的方法
10    xiaobai -> walk();
11    xiaobai -> eat();
12 }

```

2.2.4. 类是一种自定义的数据类型

我们在定义类中的属性的时候，可以定义int类型、float类型、字符串类型等等，那么能不能定义为另外的一个类的类型呢？

可以的！类其实就一种自定义的复杂的数据类型。

```

1 class Dog {
2 public:
3     string name;
4     string color;
5
6     void bark() {
7         cout << "汪汪汪" << endl;
8     }
9 };
10
11 class Person {
12 public:
13     string name;
14     string gender;
15     int age;
16     Dog dog;
17
18     void walk() {
19         cout << "人类会走路" << endl;
20     }
21
22     void eat() {
23         cout << "人类会吃饭" << endl;
24     }
25 };

```

2.2.5. 类外和其他文件中实现类函数

2.2.5.1. 类外实现

```

1 class Person {
2 public:
3     void walk();           // 在这里只是定义
4 }
5
6 void Person::walk() {
7     cout << "person walk" << endl;
8 }
```

2.2.5.2. 其他文件中实现

如果我们设计的类需要在其他的文件中访问，需要设计头文件！

person.h

```

1 #ifndef BASIC_LEARNING_PERSON_H
2 #define BASIC_LEARNING_PERSON_H
3
4 class Person {
5 public:
6     void walk();
7 };
8
9 #endif
```

person.cpp

```

1 #include "Person.h"
2
3 void Person::walk() {
4     cout << "person walk" << endl;
5 }
```

2.2.6. 静态

我们在类中定义成员的时候（函数、属性），可以使用关键字**static**来修饰，而这里的关键字**static**表示的就是静态。

在一个类中，被**static**修饰的成员，称为静态成员，可以分为：静态属性、静态函数

2.2.6.1. 静态属性

静态的属性内存是开辟在全局区的，与对象无关，并不隶属于对象。在程序编译的时候，就已经完成了空间的开辟与初始化的赋值操作了，并且在程序运行的整个过程中是始终保持的。

静态属性的空间开辟早于对象的创建，并且静态属性不隶属于对象，而是被所有的对象所共享的。因此，如果你希望某一个属性是可以被所有的对象所共享的，就可以设置为静态的属性。

```
1 #include <iostream>
2
3 using namespace std;
4
5 namespace part1 {
6     class Person
7     {
8     public:
9         // 静态的成员变量，必须在类内定义、类外初始化赋值
10        static int countOfObjs;
11        // 静态的常量，可以在类内定义，并同时进行初始化；也可以在类内定义、类外进行初始化
12        const static int MIN_AGE;
13    };
14    int Person::countOfObjs = 0;
15    const int Person::MIN_AGE = 0;
16 }
17
18 using namespace part1;
19
20 int main() {
21
22     // 访问静态成员变量（属性）
23
24     // 可以直接使用类来访问
25     Person::countOfObjs = 20;
26
27     // 也可以通过对对象来访问，但是即便使用不同的对象，访问到的空间仍然是相同的空间
28     Person xiaobai;
29     Person xiaohei;
30     xiaohei.countOfObjs = 100;
31     cout << xiaobai.countOfObjs << endl;
32
33     return 0;
34 }
```

2.2.6.2. 静态函数

被关键字static修饰的函数就是静态函数，与静态属性类似，静态函数依然不隶属于某一个对象，而是隶属于当前类的。静态的函数可以使用对象来调用，也可以直接使用类来调用。

```

1 #include <iostream>
2
3 using namespace std;
4
5 namespace part1 {
6     class Person
7     {
8         public:
9             static void test() {
10                 cout << "person test" << endl;
11             }
12     };
13 }
14
15 using namespace part1;
16
17 int main() {
18     // 使用类来访问
19     Person::test();
20
21     return 0;
22 }
```

2.3. 构造与析构

2.3.1. 构造函数的介绍

构造函数，是一个比较特殊的函数。我们在使用一个类的对象的时候，需要为其分配空间。**空间分配完成之后**，我们一般都会对创建的对象的**属性进行初始化**的操作。而这个过程就可以在构造函数中来完成了。

因此：构造函数是一个函数，是在对象创建的时候触发，用来对对象的属性进行初始化的赋值操作。

2.3.2. 构造函数的定义

- 构造函数的名字，必须和类的名字相同！
- 构造函数不能写返回值类型！
- 构造函数可以有不同的重载！

```

1 class Person {
2 public:
3     // 这就是一个无参的构造函数
4     Person() {
5         cout << "无参构造函数执行了!" << endl;
6     }
7
8     // 这就是一个有参的构造函数
9     Person(int age) {
10        cout << "有参构造函数执行了! 参数age = " << age << endl;
11    }
12 }

```

2.3.3. 构造函数的使用

```

1 // 构造函数的定义:
2 //      1、构造函数没有返回值类型, 不能写, 连void都不能写
3 //      2、构造函数的名字必须与类名相同
4 //      3、构造函数可以通过不同的参数, 来实现重载(Overload)
5
6 class Person {
7 public:
8     Person() {
9         cout << "Person类的无参构造函数执行了" << endl;
10    }
11
12     Person(int age) {
13         cout << "Person类的有参构造函数执行了" << endl;
14    }
15
16     Person(int age, int score) {
17         cout << "Person(int, int)构造函数执行了" << endl;
18    }
19 };
20
21 int main() {
22
23     // 构造函数的使用: 是在创建对象的时候调用的。
24     //
25     // 显式调用
26     // Person xiaoming = Person();
27     // Person xiaoming = Person(10);
28     // Person xiaoming = Person(10, 20);
29     //
30     // Person xiaoming;      // 注意事项: 如果用这种缩写的方式, 切记, 如果使用无参构造函数的方
31     // 式来创建对象, 不能添加()
32     // Person xiaoming(10);

```

```

32 // Person xiaoming(10, 20);
33
34 // 隐式调用
35 // Person xiaoming = {};
36 // Person xiaoming = {10};      // 这里的大括号可以省略不写
37 // Person xiaoming = {10, 20};
38
39 // Person* xiaoming = new Person();
40 // Person* xiaoming = new Person(10);
41 // Person* xiaoming = new Person(10, 20);
42
43     return 0;
44 }
```

2.3.4. explicit关键字

c++提供了关键字explicit，禁止通过构造函数进行的隐式转换。声明为explicit的构造函数不能在隐式转换中使用。

```

1 class Person {
2 public:
3     int age;
4     explicit Person(int a) {
5         age = a;
6     }
7 }
8
9 int main() {
10    // 这里创建对象会出错，因为一个参数的构造函数已经被修饰为了explicit，不允许隐式转换。
11    Person xiaoming = 10;
12
13    return 0;
14 }
```

2.3.5. 构造函数注意事项

- 如果在一个类中，没有手动写任意的构造函数，此时系统会自动为其提供一个public权限的无参构造函数。
- 如果在一个类中，写了任意的一个构造函数，此时系统将不再提供默认的无参构造函数。如果需要的话，需要自己书写。

```

1 class Person {
2 public:
3     int age;
4
5     Person(int age) {
6         this->age = age;
7     }
8 }
```

```

7     }
8 }
9
10 int main() {
11
12     Person p;    // 这样的对象创建会出错，因为现在Person类中，没有无参的构造函数。
13
14     return 0;
15 }
```

2.3.6. 构造函数初始化列表

我们自己书写构造函数，很大的一个用途就是对属性进行初始化的赋值操作，就像如下代码：

```

1 class Person {
2 public:
3     string name;
4     int age;
5     string gender;
6     int score;
7
8     Person() {
9         cout << "Person的无参构造函数执行了" << endl;
10        // 希望给属性进行初始化的赋值操作
11        name = "xiaobai";
12        age = 18;
13        gender = "male";
14        score = 99;
15    }
16
17    Person(string n, int a, string g, int s) {
18        cout << "Person的有参构造函数执行了" << endl;
19        // 希望给属性使用指定的值进行初始化
20        name = n;
21        age = a;
22        gender = g;
23        score = s;
24    }
25 }
```

在上述的代码中，无论是无参的构造函数还是有参的构造函数，我们的目的都是在创建对象的时候，为属性进行初始化的赋值操作。但是重复的这样的赋值有点麻烦，此时，C++为我们提供了初始化列表的方式，来对属性进行初始化的赋值操作。

```

1 class Person {
2 public:
3     string name;
```

```

4     int age;
5     string gender;
6     int score;
7
8     Person() : name("xiaobai"), age(18), gender("male"), score(99)
9     {
10        cout << "Person的无参构造函数执行了" << endl;
11    }
12
13    Person(string n, int a, string g, int s) : name(n), age(a), gender(g),
14    score(s)
15    {
16        cout << "Person的有参构造函数执行了" << endl;
17    }

```

2.3.7. 拷贝构造函数

拷贝构造函数是C++中的另外一种构造函数，这个构造函数也是可以由系统自动提供的。

- 1 如果我们没有给一个类写拷贝构造函数，系统会自动的生成一个默认的拷贝构造函数，为每一个属性进行赋值。
- 2 如果需要在拷贝构造函数中实现自己的拷贝逻辑，需要自己书写拷贝构造函数。

系统默认的拷贝构造函数：

```

1 class Person {
2 public:
3     string name;
4     int age;
5
6     Person() {
7         cout << "Person的无参构造函数执行了" << endl;
8     }
9
10    Person(string n, int a) {
11        name = n;
12        age = a;
13        cout << "Person的有参构造函数执行了" << endl;
14    }
15
16
17    int main() {
18
19        // 在上述的Person类中，我们并没有去写拷贝构造函数，此时系统会自动的提供一个拷贝构造函数，实现
20        // 对属性的赋值操作。
21        Person p1("xiaoming", 19);

```

```

21 // 这里就是默认执行的拷贝构造函数，相当于 Person p2 = Person(p1);
22 // 得到的对象p2的属性值与p1对象的属性值完全相同
23 Person p2(p1);
24
25 cout << "p1.name = " << p1.name << ", p1.age = " << p1.age << endl;
26 cout << "p2.name = " << p2.name << ", p2.age = " << p2.age << endl;
27
28 return 0;
29 }
```

自己实现拷贝构造函数：

```

1 class Person {
2 public:
3     string name;
4     int age;
5
6     Person() {
7         cout << "Person的无参构造函数执行了" << endl;
8     }
9
10    Person(string n, int a) {
11        name = n;
12        age = a;
13        cout << "Person的有参构造函数执行了" << endl;
14    }
15
16    Person(const Person& p) {
17        cout << "Person的拷贝构造函数执行了" << endl;
18        name = p.name;
19        age = p.age;
20    }
21 }
22
23 int main() {
24
25     Person p1("xiaoming", 19);
26     // 这里就是执行的拷贝构造函数，相当于 Person p2 = Person(p1);
27     // 得到的对象p2的属性值与p1对象的属性值完全相同
28     Person p2(p1);
29
30     cout << "p1.name = " << p1.name << ", p1.age = " << p1.age << endl;
31     cout << "p2.name = " << p2.name << ", p2.age = " << p2.age << endl;
32
33     return 0;
34 }
```

2.3.8. 析构函数

我们将一个对象从空间开辟开始，到空间销毁结束，这样的过程称为是一个对象的一生，用“生命周期”来描述这样 的过程。对象的生命周期的起点是构造函数，而对象的生命周期的终点就是析构函数。

析构函数，将会在对象被销毁之前自动调用。

析构函数也是可以由系统自动生成的。

```

1 class Person {
2 public:
3     int age;
4     int* score;
5
6     Person() {
7         cout << "Person的无参构造函数执行了" << endl;
8     }
9
10    // 这就是Person的析构函数，析构函数只能这样书写，且不能有参数
11    // 通常在析构函数中，我们会进行一些资源的释放，例如开辟的堆中的内存
12    ~Person() {
13        cout << "Person的析构函数执行了，表示这个对象即将被销毁了" << endl;
14        if (score != NULL) {
15            delete score;    // 释放对应的堆空间
16            score = NULL;   // 释放本来存储的地址，防止野指针
17        }
18    }
19 }
```

2.3.9. 深拷贝与浅拷贝的问题

深拷贝与浅拷贝是一个老生常谈的问题，在数组的部分提到过，在面向对象部分也有这两个概念。在这里我们首先需要先区分一下什么是深拷贝，什么是浅拷贝。

浅拷贝：在拷贝构造函数中，直接完成属性的赋值操作。

深拷贝：在拷贝构造函数中，创建一个新的空间，使得属性中的指针指向这个新的空间。

2.3.9.1. 浅拷贝案例

```

1 class Person {
2 public:
3     int age;
4     int* score;
5
6     Person(int a, int s) {
7         cout << "Person的有参构造函数执行了" << endl;
8         age = a;
9         score = new int(s);
10    }
```

```

11
12     Person(const Person& p) {
13         cout << "拷贝构造函数" << endl;
14         // 拷贝构造函数的默认实现，是直接进行属性值的拷贝
15         age = p.age;
16         score = p.score;
17     }
18
19     ~Person() {
20         cout << "析构函数执行了" << endl;
21         // 将score指向的堆空间销毁
22         if (score != NULL) {
23             delete score;
24             score = NULL;
25         }
26     }
27 }
28
29 int main() {
30     // 创建一个对象
31     Person p1(18, 99);
32     // 通过拷贝构造函数，拷贝出一个新的对象
33     Person p2(p1);
34
35     // 在刚才的拷贝构造函数中，属性值直接进行值的拷贝，这个过程就是一个浅拷贝
36     // 对比两个对象的score地址，完全相同
37     cout << p1.score << endl;
38     cout << p2.score << endl;
39
40     // 现在直接运行这个程序就会出问题了
41
42     // 问题出现原因：
43     // 由于现在是浅拷贝，p2的属性score和p1的属性score存储的地址是完全相同的。
44     // main函数执行结束，p1、p2都是局部变量，需要销毁。
45     // 先销毁p2，执行p2的析构函数，此时p2.score指向的空间被销毁了。
46     // 再销毁p1，执行p1的析构函数，此时p1.score指向了一个已经被销毁了空间，p1.score已经是一个野
47     // 指针了。会出现问题！
48
49     return 0;
}

```

2.3.9.2. 深拷贝案例

```

1 class Person {
2 public:
3     int age;
4     int* score;
5

```

```

6  Person(int a, int s) {
7      cout << "Person的有参构造函数执行了" << endl;
8      age = a;
9      score = new int(s);
10 }
11
12 Person(const Person& p) {
13     cout << "拷贝构造函数" << endl;
14     // 拷贝构造函数的默认实现，是直接进行属性值的拷贝
15     age = p.age;
16
17     // 这里不再是简简单单的值拷贝，而是在堆上创建了一个新的空间，新空间中存储原来p的score对应的值
18     // 然后将这个新的空间地址给score进行赋值
19     score = new int(*p.score);
20 }
21
22 ~Person() {
23     cout << "析构函数执行了" << endl;
24     // 将score指向的堆空间销毁
25     if (score != NULL) {
26         delete score;
27         score = NULL;
28     }
29 }
30 }
31
32 int main() {
33     // 创建一个对象
34     Person p1(18, 99);
35     // 通过拷贝构造函数，拷贝出一个新的对象
36     Person p2(p1);
37
38     // 在刚才的拷贝构造函数中，score属性是开辟了一个新的空间的
39     // 对比两个对象的score地址，不相同的
40     cout << p1.score << endl;
41     cout << p2.score << endl;
42
43     // 对比两个对象的score指向的值，是相同的
44     cout << *p1.score << endl;
45     cout << *p2.score << endl;
46
47     // 此时程序执行结束前，p2.score指向的空间被销毁，与p1.score指向的空间没有关系
48     // 因此，p1.score在进行空间销毁的时候也就不会有任何问题
49     return 0;
50 }

```

2.4. this指针

2.4.1. this是什么

在C++中，this是一个指针，用来指向当前的对象的！

```

1 class Person {
2 public:
3     Person() {}
4     Person(int a): age(a) {}
5
6     int getAge() {
7         return age;
8     }
9
10 private:
11     int age;
12 }
```

在上述代码中，类Person中有一个函数getAge，可以返回属性age的值。那么问题来了，一个类可以有多个对象的。而非静态的属性age是隶属于对象的。不同的对象的age，在内存中的空间肯定也是不同的。如何区分需要返回哪一个对象的age呢？

在一个类中，涉及到成员的访问的时候，非静态的成员访问，通常都会使用this指针来访问。

```

1 class Person {
2 public:
3     Person() {}
4     Person(int a): age(a) {}
5
6     // 这里使用this指针来访问age属性，这种写法是默认的，并且this是可以省略不写的
7     // 那么所谓的当前对象到底是谁呢？
8     // 最简单的理解就是：哪个对象调用这个函数，this指针就指向谁
9     int getAge() {
10         return this->age;
11     }
12
13 private:
14     int age;
15 }
```

2.4.2. this不可省略的情况

绝大多数的情况下，在一个类的内部，访问当前类中的非静态成员的时候，this指针都是可以省略不写的。但是有一种情况，this指针不能省略，必须要显式的写出来：

如果在一个函数中出现了与当前类的属性同名字的局部变量！为了区分局部变量还是属性，此时的this指针不能省略。

```

1 class Person {
2 public:
3     int age;
4
5     // 在这个构造函数中，出现了一个局部变量age，与属性名字相同了
6     // 于是为了区分这两个，需要使用this指针显式的指向age属性来进行访问
7     Person(int age) {
8         this->age = age;
9     }
10 }
```

2.4.3. 返回当前对象的函数

```

1 class Point {
2 public:
3     int x;
4     int y;
5     Point() {}
6     Point(int x, int y): x(x), y(y) {}
7
8     // 返回当前对象的引用
9     Point& add(int deltaX, int deltaY) {
10         x += deltaX;
11         y += deltaY;
12         // this是一个指针，用来指向当前的对象
13         // 因此，如果需要返回当前的对象的话，就需要使用*来访问
14         return *this;
15     }
16 }
```

2.4.4. 空指针访问成员函数

在C++中，使用空指针是可以访问成员函数的，但是需要注意：不能在函数中出现this指针！

```

1 class Person {
2 public:
3     int age;
4
5     Person() {}
6     Person(int age): age(age) {}
7
8     void testFunction01() {
9         cout << "testFunction01执行了" << endl;
10    }
11 }
```

```

12 void testFunction02() {
13     if (this == NULL) {
14         cout << "this是一个空指针" << endl;
15         return;
16     }
17     cout << "testFunction02执行了" << endl;
18 }
19
20 void getAge() {
21     return age;
22 }
23 }
24
25 int main() {
26     // 创建一个空对象
27     Person* person = NULL;
28
29     // 函数访问
30     person->testFunction01();           // 可以正常访问，因为在这个函数中没有使用到this指针
31     person->testFunction02();           // 访问出问题，因为在这个函数中使用到了this指针
32     person->getAge();                 // 访问出问题，在这个函数中虽然没有写this，但是有属性的
33     // 访问，默认就是用到的this
34
35     return 0;
}

```

2.4.5. 常函数与常对象

2.4.5.1. 什么是常函数

- 使用关键字const修饰的函数，叫做常函数。
- 常函数中，不允许修改属性的值。
- 常函数中，不允许调用其他的普通函数。
- 如果想要在常函数中修改某个属性的值，需要将这个属性设置为mutable。

```

1 class Person {
2 public:
3     string name;
4     int age;
5     mutable int score;      // 修饰为可变的，这个属性可以在常函数中进行修改
6
7     Person(string name, int age, int score): name(name), age(age), score(score) {}
8
9     // 定义常函数
10    void fixPerson(string newName, int newAge, int newScore) const {
11        name = newName;       // 这里会出错，不允许在常函数中修改普通属性的值
12        age = newAge;        // 这里会出错，不允许在常函数中修改普通属性的值
13        score = newScore;     // 这里可以正常修改，因此此时的score已经被修饰为mutable
}

```

```

14         test();      // 这里会出错，不允许在常函数中调用其他的普通函数
15     }
16
17
18     void test() {}
19 }
```

2.4.5.2. 常对象

- 在对象创建的时候，使用const修饰的对象，就是常对象。
- 常对象可以访问任意的属性值，但是不能修改普通属性的值。
- 常对象可以修改mutable属性的值。
- 常对象只能调用常函数。

```

1 int main() {
2     // 创建一个常对象
3     const Person person("zhangsan", 18, 99);
4
5     // 使用这个常对象进行属性访问
6     cout << person.name << endl;           // 可以正常访问
7     person.age = 100;                      // 不能进行修改
8     person.score = 200;                    // mutable属性，可以修改
9 }
```

2.5. 友元

2.5.1. 友元是什么

类的主要特点之一是数据隐藏，即类的私有成员无法在类的外部（作用域之外）访问。但是，有时候需要在类的外部访问类的私有成员，怎么办？

解决方法是使用友元函数，友元函数是一种特权函数，C++允许这个特权函数访问私有成员。这一点从现实生活中也可以很好的理解：

比如你的家，有客厅，有你的卧室，那么你的客厅是Public的，所有来的客人都可以进去，但是你的卧室是私有的，也就是说只有你能进去，但是呢，你也可以允许你的闺蜜好基友进去。

程序员可以把一个全局函数、某个类中的成员函数、甚至整个类声明为友元。

2.5.2. 全局函数做友元

```

1 class Home {
2     // 将全局函数作为友元
3     friend void gotoBed(Home* home);
4
5 public:
```

```

6     string livingRoom = "这里是客厅";
7
8 private:
9     string bedRoom = "这里是卧室";
10 }
11
12 void gotoBed(Home *home) {
13     // 可以访问公共部分
14     cout << home->livingRoom << endl;
15     // 可以访问私有部分
16     cout << home->bedRoom << endl;
17 }
18
19 int main() {
20
21     // 创建对象
22     Home home;
23     gotoBed(&home);
24
25     return 0;
26 }
```

2.5.3. 成员函数做友元

```

1 // 定义有这样一个类，但是类中的成员是看不到的
2 class Home;
3
4 // 好基友
5 class GoodGuy {
6 public:
7     GoodGuy();
8     void visit();
9 private:
10    Home* home;
11 };
12
13 // 我的家
14 class Home {
15     // 将基友的visit函数作为友元
16     friend void GoodGuy::visit();
17 public:
18     string livingRoom = "客厅";
19 private:
20     string bedRoom = "卧室";
21 };
22
23 GoodGuy::GoodGuy() {
24     this->home = new Home();
```

```
25 }
26
27 void GoodGay::visit() {
28     cout << home->livingRoom << endl;
29     cout << home->bedRoom << endl;
30 }
31
32 int main() {
33
34     GoodGay gay;
35     gay.visit();
36
37     return 0;
38 }
```

2.5.4. 类做友元

```
1 class Home;
2
3 // 友元类
4 class GoodGay {
5 public:
6     GoodGay();
7     void visit();
8
9 private:
10    Home* home;
11 };
12
13 class Home {
14     // 友元类
15     friend class GoodGay;
16 public:
17     string livingRoom = "客厅";
18 private:
19     string bedRoom = "卧室";
20 };
21
22 GoodGay::GoodGay() {
23     home = new Home();
24 }
25 void GoodGay::visit() {
26     cout << home->livingRoom << endl;
27     cout << home->bedRoom << endl;
28 }
29
30 int main() {
```

```

32     GoodGay gay;
33     gay.visit();
34
35     return 0;
36 }
```

2.6. 运算符重载

2.6.1. 什么是运算符重载

运算符重载，就是对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。

运算符重载(operator overloading)只是一种“语法上的方便”，也就是它只是另一种函数调用的方式。

在c++中，可以定义一个处理类的新运算符。这种定义很像一个普通的函数定义，只是函数的名字由关键字operator及其紧跟的运算符组成。差别仅此而已。它像任何其他函数一样也是一个函数，当编译器遇到适当的模式时，就会调用这个函数。

语法：

定义重载的运算符就像定义函数，只是该函数的名字是operator@，这里的@代表了被重载的运算符。函数的参数中参数个数取决于两个因素。

运算符是一元(一个参数)的还是二元(两个参数)；

运算符被定义为全局函数(对于一元是一个参数，对于二元是两个参数)还是成员函数(对于一元没有参数，对于二元是一个参数-此时该类的对象用作左耳参数)

注意：

有些人很容易滥用运算符重载。它确实是一个有趣的工具。但是应该注意，它仅仅是一种语法上的方便而已，是另外一种函数调用的方式。从这个角度来看，只有在能使涉及类的代码更易写，尤其是更易读时(请记住，读代码的机会比我们写代码多多了)才有理由重载运算符。如果不是这样，就改用其他更易用，更易读的方式。

2.6.2. 可重载的运算符

几乎所有的运算符都可以重载，但运算符重载的使用时相当受限制的。特别是不能改变运算符优先级，不能改变运算符的参数个数。这样的限制有意义，否则，所有这些行为产生的运算符只会混淆而不是澄清语意。

可以重载的操作符

+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%
^=	&=	=	<<	>>	>>=	<<=	=	!=
<=	>=	&&		++	--	->*	'	->
[]	0	new	delete	new[]	delete[]			

不能重载的算符

. :: .* ?: **sizeof**

2.6.3. 运算符重载： +

```

1 class Point {
2     public:
3         // 定义属性
4         int x;
5         int y;
6
7         // 定义构造函数，用来初始化属性
8         Point(): x(0), y(0) {}
9         Point(int x, int y): x(x), y(y) {}
10
11        // 在类内实现的运算符重载
12        Point operator+(const Point& p) const {
13            return {x + p.x, y + p.y};
14        }
15    };
16
17    // 全局函数实现运算符重载
18    Point operator-(const Point& p1, const Point& p2) {
19        return {p1.x - p2.x, p1.y - p2.y};
20    }
21
22    int main() {
23
24        Point p1(10, 20);
25        Point p2(15, 25);
26
27        Point res = p1 + p2;

```

```

28     cout << "res.x = " << res.x << ", res.y = " << res.y << endl;
29
30     Point res2 = p1 - p2;
31     cout << "res2.x = " << res2.x << ", res2.y = " << res2.y << endl;
32
33     return 0;
34 }
```

2.6.4. 运算符重载：++

```

1 class Point {
2 public:
3     // 定义属性
4     int x;
5     int y;
6
7     // 定义构造函数，用来初始化属性
8     Point(): x(0), y(0) {}
9     Point(int x, int y): x(x), y(y) {}
10
11    // 运算符前置，先运算、后取值
12    Point operator++() {
13        x++;
14        y++;
15        return *this;
16    }
17
18    // 在类内实现的运算符重载，运算符后置
19    Point operator++(int) {
20        // 先创建一个对象，记录原来的值
21        Point tmp = *this;
22        // 属性自增
23        x++;
24        y++;
25        // 返回之前记录的值
26        return tmp;
27    }
28 };
29
30 Point operator--(Point& point) {
31     point.x--;
32     point.y--;
33     return point;
34 }
35
36 Point operator--(Point& point, int) {
37     Point tmp = point;
38     point.x--;
```

```

39     point.y--;
40     return tmp;
41 }
42
43 int main() {
44
45     Point p1(10, 20);
46     Point p2(15, 25);
47
48     Point res1 = ++p1;
49     cout << "res1.x = " << res1.x << ", res1.y = " << res1.y << endl;
50     cout << "p1.x = " << p1.x << ", p1.y = " << p1.y << endl;
51
52     Point res2 = p2++;
53     cout << "res2.x = " << res2.x << ", res2.y = " << res2.y << endl;
54     cout << "p2.x = " << p2.x << ", p2.y = " << p2.y << endl;
55
56     return 0;
57 }
```

2.6.5. 运算符重载：<<

```

1 class Point {
2     friend ostream& operator<<(ostream& os, const Point& p);
3 public:
4     // 定义属性
5     int x;
6     int y;
7
8     // 定义构造函数，用来初始化属性
9     Point(): x(0), y(0), privateField(0) {}
10    Point(int x, int y): x(x), y(y), privateField(0) {}
11
12 private:
13     int privateField;
14 };
15
16 // 在类外定义运算符重载，全局函数
17 // 我希望在这里能够将Point类中的私有属性也拼接起来，因此需要做成友元
18 ostream& operator<<(ostream& os, const Point& p) {
19     os << "x = " << p.x << ", y = " << p.y << ", privateField = " <<
20     p.privateField;
21     return os;
22 }
23
24 int main() {
25     Point p1(10, 20);
26     cout << "p1: " << p1 << endl;
```

```

26
27     return 0;
28 }
```

2.6.6. 运算符重载： =

```

1 class Person {
2 public:
3     int age;
4     int score;
5     int* p;
6
7     Person(): age(0), score(0), p(nullptr) {}
8
9     Person(const Person& person) {
10        cout << "拷贝构造函数执行了" << endl;
11        age = person.age;
12        score = person.score;
13        p = new int(*person.p);
14    }
15
16 // 重载赋值运算符
17 Person& operator=(const Person& person) {
18    cout << "重载赋值运算符执行了" << endl;
19    age = person.age;
20    score = person.score;
21    p = new int(*person.p);
22    return *this;
23 }
24
25 ~Person() {
26    cout << "析构函数执行了" << endl;
27    if (p == nullptr) {
28        delete p;
29        p = nullptr;
30    }
31 }
32 };
33
34 int main() {
35
36     // 创建一个对象
37     Person p1;
38     p1.age = 18;
39     p1.score = 99;
40     p1.p = new int(100);
41 }
```

```

42 // 在这里，虽然是等号运算符，但是p2对象还没有完成空间开辟、实例化，那么在这里会调用拷贝构造函数，而非重载的等号运算符
43 Person p2 = p1;
44
45 // 修改p2的属性值
46 p2.age = 20;
47 p2.score = 100;
48 p2.p = new int(200);
49
50 // 这里的p1已经开辟空间了，这里就会触发重载的等号运算符
51 p1 = p2;
52 cout << p1.age << ", " << p1.score << ", " << p1.p << " =>" << *p1.p << endl;
53 cout << p2.age << ", " << p2.score << ", " << p2.p << " =>" << *p2.p << endl;
54
55
56 return 0;
57 }

```

2.7. 封装

面向对象编程思想中，有三大特性：封装、继承、多态。

封装可以有广义和狭义上的概念。广义上的封装，我们可以将一些功能相近的一些类放入一个模块中。这里我们更多强调的是狭义上的封装性。

- 定义：我们可以通过对具体属性的封装实现，把对成员变量的访问进行私有化，让他只能在类内部可见，通过公共的方法间接实现访问。
- 优点：提高了代码的安全性，复用性和可读性。

```

1 class Student {
2     // 将不希望对外提供直接访问的属性封装起来
3 private:
4     string name;
5     int age;
6
7 public:
8     Student(): name(""), age(0) {}
9     Student(string name, int age) {}
10
11    void setName(string name) {
12        this->name = name;
13    }
14    void getName() {
15        return name;
16    }
17
18    void setAge(int age) {
19        if (age >= 0 && age <= 130) {

```

```

20         this->age = age;
21     }
22 }
23 int getAge() {
24     return age;
25 }
26 }
```

2.8. 继承

2.8.1. 程序中的继承

在现实生活中，我们与父母有继承的关系，在java中也存在继承的思想，来提高代码的复用性、代码的拓展性。

程序中的继承，是类与类之间的特征和行为的一种赠予或获取。一个类可以将自己的属性和方法赠予其他的类，一个类也可以从其他的类中获取他们的属性和方法。

两个类之间的继承，必须满足 **is a** 的关系。

两个类之间，A类将属性和特征赠予B类。此时A类被称为是父类，B类被称为是子类，两者之间的关系是子类继承自父类



2.8.2. 继承的语法

在C++中，在定义类的时候，类名后面使用冒号来定义父类。

- 类中的所有成员都可以继承给子类，但是私有的成员，由于访问权限的限制，子类无法访问。
- 一个类在继承了其他类之后，也可以被其他类继承。
- 使用继承，可以简化代码、提高代码的复用性、提高代码的拓展性，最重要的是让类与类之间产生了继承关系，是多态的前提。

```

1 class Animal {
2 public:
3     int age;
4 }
5 class Dog: public Animal {
```

```

6     void test() {
7         cout << age << endl;
8     }
9 }
10 int main() {
11     Dog dog;
12     dog.test();
13
14     return 0;
15 }
```

2.8.3. 继承的三种方式

在C++中，继承有三种方式，分别是：公共继承、保护继承和私有继承。其实只是一个访问权限的问题。

- **公共继承：**继承到父类中的属性，保留原本的访问权限（私有除外）
- **保护继承：**继承到父类中的属性，超过protected权限的部分将降为protected权限（私有除外）
- **私有继承：**继承到父类中的属性，访问权限都为private权限（私有除外）

C++中默认使用的是私有继承！

2.8.3.1. 公共继承

```

1 // 定义类，分别定义三种访问权限的属性
2 class BaseClass {
3 public:
4     int publicField;
5 protected:
6     int protectedField;
7 private:
8     int privateField;
9 }
10 // 定义自类，公共继承
11 // 继承到的 publicField，还是public权限
12 // 继承到的 protectedField，还是protected权限
13 // privateField是私有权限的，无法继承给自类
14 class SubClass: public BaseClass {
15     void test() {
16         cout << publicField << endl;
17         cout << protectedField << endl;
18     }
19 }
20
21 int main() {
22     SubClass sc;
23     cout << sc.publicField << endl;      // 在类外依然可以访问
24     // cout << sc.protectedField << endl;    // 类外不能访问，因为这是保护权限
```

25 | }

2.8.3.2. 保护继承

```

1 // 定义类，分别定义三种访问权限的属性
2 class BaseClass {
3 public:
4     int publicField;
5 protected:
6     int protectedField;
7 private:
8     int privateField;
9 }
10 // 定义自类，保护继承
11 // 继承到的 publicField，原来是public权限，现在是protected权限
12 // 继承到的 protectedField，还是protected权限
13 // privateField是私有权限的，无法继承给自类
14 class SubClass: protected BaseClass {
15     void test() {
16         cout << publicField << endl;
17         cout << protectedField << endl;
18     }
19 }
20
21 int main() {
22     SubClass sc;
23     // cout << sc.publicField << endl;      // 在类不能访问，因为这是保护权限
24     // cout << sc.protectedField << endl;    // 类外不能访问，因为这是保护权限
25 }
```

2.8.3.3. 私有继承

```

1 // 定义类，分别定义三种访问权限的属性
2 class BaseClass {
3 public:
4     int publicField;
5 protected:
6     int protectedField;
7 private:
8     int privateField;
9 }
10 // 定义自类，私有继承
11 // 继承到的 publicField，是私有权限
12 // 继承到的 protectedField，是私有权限
13 // privateField是私有权限的，无法继承给自类
14 class SubClass: public BaseClass {
15     void test() {
16         cout << publicField << endl;
17         cout << protectedField << endl;
```

```

18     }
19 }
20
21 int main() {
22     SubClass sc;
23     // cout << sc.publicField << endl;      // 类外不能访问，因为这里是私有权限
24     // cout << sc.protectedField << endl;    // 类外不能访问，因为这里是私有权限
25 }
```

2.8.4. 继承中的构造和析构

子类对象在创建的时候，需要先调用父类中的构造函数，用来初始化父类部分。因此，子类对象创建的时候，先调用父类的构造函数，再调用子类自己的构造函数。

而析构函数的调用正好相反，先调用子类的析构函数，再调用父类的析构函数。

```

1 class Animal {
2 public:
3     int age;
4
5     Animal() {
6         age = 0;
7         cout << "父类中的无参构造函数调用了" << endl;
8     }
9
10    explicit Animal(int age): age(age) {
11        cout << "父类中的有参构造函数调用了" << endl;
12    }
13
14    ~Animal() {
15        cout << "父类中的析构函数调用了" << endl;
16    }
17 };
18
19 class Dog: public Animal {
20 public:
21     explicit Dog() {
22         cout << "子类中的无参构造函数被调用了" << endl;
23     }
24
25     ~Dog() {
26         cout << "子类中的析构函数被调用了" << endl;
27     }
28 };
29
30 int main() {
31     Dog dog();
```

```

33
34     return 0;
35 }
```

从上述的代码中可以看到，子类对象在创建的时候，需要先调用父类中的构造函数来构造父类部分。这里默认是调用父类中的无参构造函数。那么问题来了：如果父类中没有无参构造函数，或者父类中的无参构造函数是私有的，怎么办？

```

1 class Animal {
2 public:
3     int age;
4
5     explicit Animal(int age): age(age) {
6         cout << "父类中的有参构造函数调用了" << endl;
7     }
8
9     ~Animal() {
10        cout << "父类中的析构函数调用了" << endl;
11    }
12 };
13
14 class Dog: public Animal {
15 public:
16     explicit Dog(int age) : Animal(age) {
17         cout << "子类中的无参构造函数被调用了" << endl;
18     }
19
20     ~Dog() {
21         cout << "子类中的析构函数被调用了" << endl;
22     }
23 };
24
25 int main() {
26
27     Dog dog(0);
28
29     return 0;
30 }
```

2.8.5. 父类子类成员同名的情况

如果父类和子类中出现了同名字的成员（属性、函数），子类会将从父类继承到的成员隐藏起来。此时使用子类对象来访问的时候，默认访问的是子类中的成员。如果想要访问父类中的成员，需要手动指定作用域。

```

1 class Animal {
2 public:
```

```

3     int age = 10;
4
5     void showAge() const {
6         cout << "父类中的函数showAge被调用, age = " << age << endl;
7     }
8
9 };
10
11 class Dog: public Animal {
12 public:
13     int age;
14
15     void showAge() const {
16         cout << "子类中的函数showAge被调用, age = " << age << endl;
17     }
18 };
19
20 int main() {
21
22     Dog dog;
23     dog.age = 2;
24
25     dog.showAge();           // 默认调用的是子类中的函数
26     dog.Animal::showAge();  // 如果想要调用父类中的函数, 需要显式调用
27
28     cout << dog.age << endl;          // 默认访问自类中的属性age
29     cout << dog.Animal::age << endl;    // 如果想要调用父类中的属性age, 需要显式调用
30
31     return 0;
32 }
```

2.8.6. 多继承

2.8.6.1. 多继承语法

我们可以从一个类继承，我们也可以同时从多个类继承，这就是多继承。但是由于多继承是非常受争议的，从多个类继承可能会导致函数、变量等同名导致较多的歧义。

多继承会带来一些二义性的问题，如果两个基类中有同名的函数或者变量，那么通过派生类对象去访问这个函数或变量时就不能明确到底调用从基类1继承的版本还是从基类2继承的版本？

解决方法就是显示指定调用那个基类的版本。

```

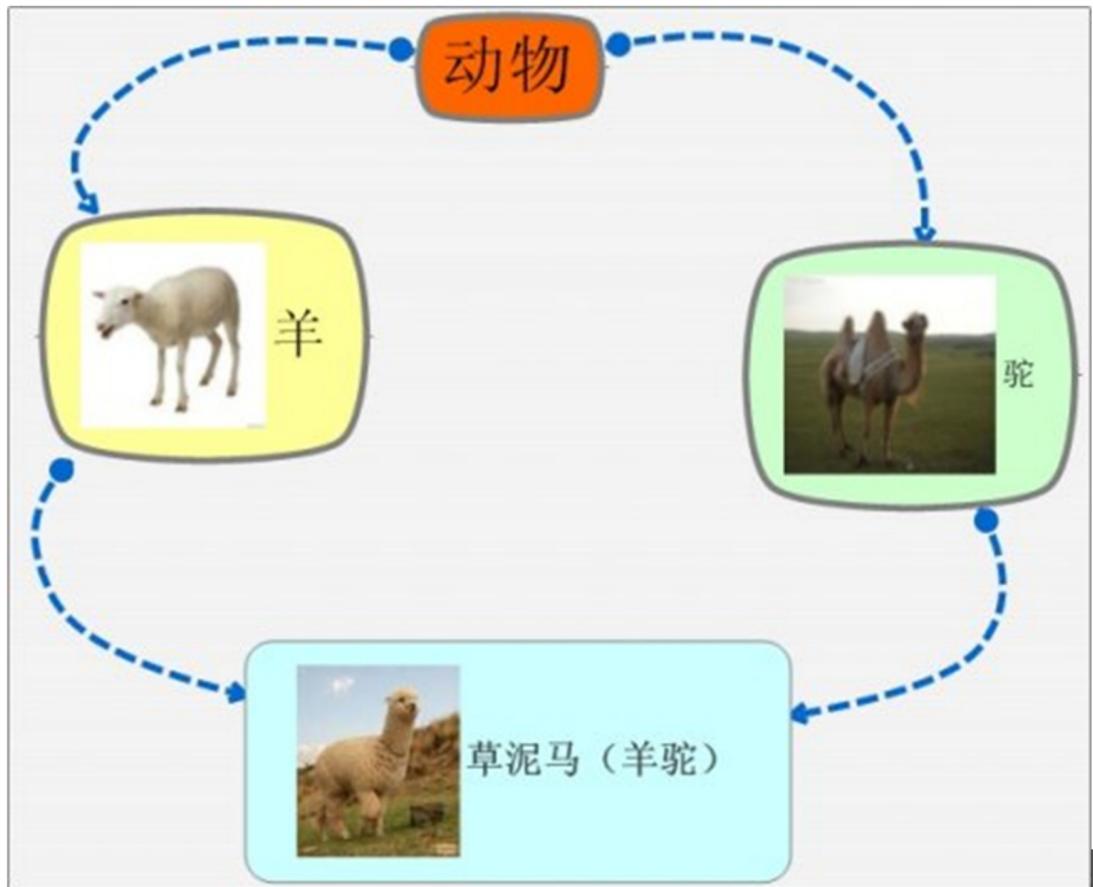
1 class Base1{
2 public:
3     void func1() { cout << "Base1::func1" << endl; }
4 }
5 class Base2{
6 public:
```

```

7     void func1() { cout << "Base1::func1" << endl; }
8     void func2() { cout << "Base2::func2" << endl; }
9 };
10
11 // 派生类继承Base1、Base2
12 class Derived : public Base1, public Base2{};
13 int main(){
14
15     Derived derived;
16     // func1是从Base1继承来的还是从Base2继承来的?
17     // derived.func1();
18     derived.func2();
19
20     //解决歧义:显示指定调用那个基类的func1
21     derived.Base1::func1();
22     derived.Base2::func1();
23
24     return 0;
25 }
```

2.8.6.2. 菱形继承

两个派生类继承同一个基类而又有某个类同时继承者两个派生类，这种继承被称为菱形继承，或者钻石型继承。



这种继承所带来的问题：

- 羊继承了动物的数据和函数，驼同样继承了动物的数据和函数，当草泥马调用函数或者数据时，就会产生二义性。

- 草泥马继承自动物的函数和数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以。

```

1 class BigBase{
2 public:
3     BigBase(){ mParam = 0; }
4     void func(){ cout << "BigBase::func" << endl; }
5 public:
6     int mParam;
7 };
8
9 class Base1 : public BigBase {
10};
11
12 class Base2 : public BigBase {
13};
14
15 class Derived : public Base1, public Base2 {
16};
17
18
19 int main(){
20
21     Derived derived;
22     // 对"func"的访问不明确
23     // derived.func();
24     // cout << derived.mParam << endl;
25     cout << "derived.Base1::mParam:" << derived.Base1::mParam << endl;
26     cout << "derived.Base2::mParam:" << derived.Base2::mParam << endl;
27
28     return 0;
29 }
```

2.8.6.3. 虚继承

Base1, Base2采用虚继承方式继承BigBase,那么BigBase被称为虚基类。

通过虚继承解决了菱形继承所带来的二义性问题。

```

1 class BigBase{
2 public:
3     BigBase(){ mParam = 0; }
4     void func(){ cout << "BigBase::func" << endl; }
5 public:
6     int mParam;
7 };
8
9 class Base1 : virtual public BigBase{};
10 class Base2 : virtual public BigBase{};
```

```

11 class Derived : public Base1, public Base2{};
12
13 int main(){
14
15     Derived derived;
16     //二义性问题解决
17     derived.func();
18     cout << derived.mParam << endl;
19     //输出结果:12
20     cout << "Derived size:" << sizeof(Derived) << endl;
21
22     return EXIT_SUCCESS;
23 }
```

2.9. 多态

2.9.1. 多态的基本概念

2.9.1.1. 什么是多态

生活中的多态，是指的客观的事物在人脑中的主观体现。例如，在路上看到一只哈士奇，你可以看做是哈士奇，可以看做是狗，也可以看做是动物。主观意识上的类别，与客观存在的事物，存在 `is a` 的关系的时候，即形成了多态。

在程序中，一个类的引用指向另外一个类的对象，从而产生多种形态。当二者存在直接或者间接的继承关系时，父类引用指向子类的对象，即形成多态。

多态是面向对象三大特性之一，记住继承是多态的前提，如果类与类之间没有继承关系，也不会存在多态。

2.9.1.2. 多态的分类

c++支持编译时多态(静态多态)和运行时多态(动态多态)，运算符重载和函数重载就是编译时多态，而派生类和虚函数实现运行时多态。

静态多态和动态多态的区别就是函数地址是早绑定(静态联编)还是晚绑定(动态联编)。如果函数的调用，在编译阶段就可以确定函数的调用地址，并产生代码，就是静态多态(编译时多态)，就是说地址是早绑定的。而如果函数的调用地址不能编译不能在编译期间确定，而需要在运行时才能决定，这这就属于晚绑定(动态多态，运行时多态)。

2.9.2. 对象转型

对象可以作为自己的类或者作为它的基类的对象来使用。还能通过基类的地址来操作它。取一个对象的地址(指针或引用)，并将其作为基类的地址来处理，这种称为向上类型转换。

也就是说：父类引用或指针可以指向子类对象，通过父类指针或引用来操作子类对象。

```

1 class Animal {
2 public:
3     void bark() {
4         cout << "Animal Bark" << endl;
```

```

5     }
6 };
7
8 class Dog: public Animal {
9 public:
10    void bark() {
11        cout << "Dog Bark" << endl;
12    }
13};
14
15 int main(){
16
17    // 父类的引用指向子类的对象
18    Dog dog;
19    Animal& animal = dog;
20
21    // 向上转型后的对象调用父类中的函数
22    animal.bark();
23
24    return 0;
25}

```

2.9.3. 虚函数

上述代码的运行结果是：Animal bark。说明执行的是父类中的bark函数，而非子类中的函数。

于是现在就有一个问题出现了：为什么？animal的引用指向的实际上是一个Dog对象，但是为什么会调用父类中的函数实现呢？

解决这个问题，我们需要了解下绑定(捆绑, binding)概念。

把函数体与函数调用相联系称为绑定(捆绑, binding)

当绑定在程序运行之前(由编译器和连接器)完成时，称为早绑定(early binding)。

上面的问题就是由于早绑定引起的，因为编译器在只有Animal地址时并不知道要调用的正确函数。编译是根据指向对象的指针或引用的类型来选择函数调用。这个时候由于调用函数的时候使用的是Animal类型，编译器确定了应该调用的bark是Animal::bark的，而不是真正传入的对象Dog::bark。

解决方法就是迟绑定(迟捆绑, 动态绑定, 运行时绑定, late binding)，意味着绑定要根据对象的实际类型，发生在运行。

C++语言要实现这种动态绑定，必须有某种机制来确定运行时对象的类型并调用合适的成员函数。对于一种编译语言，编译器并不知道实际的对象类型（编译器并不知道Animal类型的指针或引用指向的实际的对象类型）

C++动态多态性是通过虚函数来实现的，虚函数允许子类（派生类）重新定义父类（基类）成员函数，而子类（派生类）重新定义父类（基类）虚函数的做法称为覆盖(override)，或者称为重写。对于特定的函数进行动态绑定，C++要求在基类中声明这个函数的时候使用virtual关键字，动态绑定也就对virtual函数起作用。

- 为创建一个需要动态绑定的虚成员函数，可以简单在这个函数声明前面加上virtual关键字，定义时候不需要.
- 如果一个函数在基类中被声明为virtual，那么在所有派生类中它都是virtual的.
- 在派生类中virtual函数的重定义称为重写(override).
- virtual关键字只能修饰成员函数.
- 构造函数不能为虚函数

```

1 class Animal {
2 public:
3     // 将需要动态绑定的函数定义为虚函数
4     virtual void bark() {
5         cout << "Animal Bark" << endl;
6     }
7 };
8
9 class Dog: public Animal {
10 public:
11     // 在子类中重写虚函数
12     void bark() override {
13         cout << "Dog Bark" << endl;
14     }
15 };
16
17 int main(){
18
19     // 将Dog对象转成父类的对象
20     // 这里是向上转型
21     Dog dog;
22     Animal& animal = dog;
23
24     // 向上转型后的对象调用父类中的函数
25     animal.bark();
26
27     return 0;
28 }
```

2.9.4. 多态案例

2.9.4.1. 未使用多态实现

```

1
2 class SF {
3     void sendPackage() {
4         cout << "SF快递为你快速发送包裹" << endl;
5     }
6 };
7
8 class EMS {
```

```

9     void sendPackage() {
10        cout << "EMS快递为您发送包裹，哪里都能送到！" << endl;
11    }
12 };
13
14 class JDL {
15     void sendPackage() {
16        cout << "JDL快递为您发送包裹，最快当日可达！" << endl;
17    }
18 };
19
20 void sendPackage(string company) {
21     if (company == "SF") {
22         new SF().sendPackage();
23     }
24     else if (company == "EMS") {
25         new EMS().sendPackage();
26     }
27     else if (company == "JDL") {
28         new JDL().sendPackage();
29     }
30     // 这里违背了程序设计原则中的开闭原则
31     // 开闭原则：对拓展开放、对修改关闭，意思是当有新的功能增加对时候，直接拓展模块，而不是修改现有的实现部分。
32 }

```

2.9.4.2. 使用多态实现

```

1 // 快递公司类
2 class ExpressCompany {
3 public:
4     virtual void sendPackage() {
5         cout << "快递公司发送包裹" << endl;
6     }
7 };
8
9 class SF: public ExpressCompany {
10    void sendPackage() override {
11        cout << "SF快递为你快速发送包裹" << endl;
12    }
13 };
14
15 class EMS: public ExpressCompany {
16    void sendPackage() override {
17        cout << "EMS快递为您发送包裹，哪里都能送到！" << endl;
18    }
19 };
20

```

```

21 class JDL: public ExpressCompany {
22     void sendPackage() override {
23         cout << "JDL快递为您发送包裹，最快当日可达！" << endl;
24     }
25 };
26
27 void sendPackage(ExpressCompany& express) {
28     express.sendPackage();
29 }
30
31 int main(){
32
33     sendPackage(*new SF);
34     sendPackage(*new EMS);
35     sendPackage(*new JDL);
36
37     return 0;
38 }
```

2.9.5. 纯虚函数与抽象类

在设计程序时，常常希望基类仅仅作为其派生类的一个接口。这就是说，仅想对基类进行向上类型转换，使用它的接口，而不希望用户实际的创建一个基类的对象。同时创建一个纯虚函数允许接口中放置成员原函数，而不一定要提供一段可能对这个函数毫无意义的代码。

- 1 例如：
- 2 我们可以设计一个交通工具类，提供最基础的运输的功能。我们在使用到交通工具的时候，往往并不是寻求交通工具的对象，而是寻求的交通工具子类的对象，例如公交车、例如地铁、例如共享单车等。而我们需要的其实是在这些子类中的运输功能实现。因此，父类交通工具类中的运输功能怎么去实现没有意义。

纯虚函数使用virtual来修饰一个函数，并且实现部分直接设置为0。

```
1 virtual void test() = 0;
```

如果一个类中包含了纯虚函数，那么这个类也自动的编程了抽象类了。抽象类无法实例化对象，并且子类必须重写实现父类中的纯虚函数，否则子类也是抽象类。

```

1 class TrafficTools {
2 public:
3     // 定义纯虚函数，此时的类是抽象类
4     virtual void transport() = 0;
5 };
6
7 class Bus: public TrafficTools {
8 public:
9     void transport() override {
10         cout << "公交车运输乘客" << endl;
```

```

11     }
12 };
13
14 class Subway: public TrafficTools {
15 public:
16     void transport() override {
17         cout << "地铁运输乘客" << endl;
18     }
19 };
20
21 void useTrafficTools(TrafficTools& trafficTools) {
22     trafficTools.transport();
23 }
24
25 int main() {
26     // 抽象类无法实例化对象
27     // new TrafficTools;
28
29     useTrafficTools(*new Bus);
30     useTrafficTools(*new Subway));
31     return 0;
32 }
```

2.9.6. 纯虚函数与多继承

多继承带来了一些争议，但是接口继承可以说一种毫无争议的运用了。

绝大多数面向对象语言都不支持多继承，但是绝大多数面向对象语言都支持接口的概念，c++中没有接口的概念，但是可以通过纯虚函数实现接口。

接口类中只有函数原型定义，没有任何数据定义。

多重继承接口不会带来二义性和复杂性问题。接口类只是一个功能声明，并不是功能实现，子类需要根据功能说明定义功能实现。

注意：除了析构函数外，其他声明都是纯虚函数。

```

1 // 定义一个功能集合，厨师类
2 class Cooker {
3 public:
4     virtual void buyFood() = 0;
5     virtual void cook() = 0;
6     virtual void eat() = 0;
7 };
8 // 定义一个功能集合，保姆类
9 class Maid {
10 public:
11     virtual void cook() = 0;
12     virtual void clean() = 0;
13     virtual void wash() = 0;
14 };
```

```

15 // 使得人类同时具备这两种功能
16 class Person: public Cooker, public Maid {
17 public:
18     // 对继承到的纯虚函数进行实现
19     void buyFood() override { cout << "买菜" << endl; }
20     void cook() override { cout << "做饭" << endl; }
21     void eat() override { cout << "吃饭" << endl; }
22     void clean() override { cout << "扫地" << endl; }
23     void wash() override { cout << "洗衣服" << endl; }
24 };
25
26
27
28 int main() {
29
30     Person xiaoming;
31     xiaoming.buyFood();
32     xiaoming.cook();
33     xiaoming.eat();
34     xiaoming.wash();
35     xiaoming.clean();
36
37     return 0;
38 }
```

2.9.7. 虚析构函数

析构函数是对象生命周期的终点，在对象被销毁之前调用。在析构函数中，我们一般会进行资源的释放、空间的销毁的操作。例如，在一个类中有指向堆空间内存的指针，我们需要通过这样的指针来销毁对应的堆空间。但是，在多态中，父类的引用可以指向子类的对象，那么我们在使用父类的引用来销毁空间的话，就有可能会出现子类中引用的堆空间无法销毁的情况，造成内存泄漏。而解决方案就是为父类添加虚析构函数。

```

1 class Animal {
2 public:
3     virtual ~Animal() {
4         cout << "父类的析构函数执行了" << endl;
5     }
6 };
7
8 class Person: public Animal {
9 public:
10    int* n;
11
12    Person() {
13        n = new int(10);
14    }
15
16    ~Person() override {
```

```

17     cout << "子类的析构函数执行了" << endl;
18     if (n != nullptr) {
19         delete n;
20         n = nullptr;
21     }
22 }
23 }
24
25 int main() {
26
27     Animal* animal = new Person();
28     // 如果没有虚析构函数的话，这里通过animal来销毁空间，的确可以把Person开辟的堆空间给销毁掉
29     // 但是，由于只会触发父类中的析构函数，因此无法将Person属性n开辟的堆空间给销毁掉，造成内存泄漏
30     // 解决方案：将父类的析构函数作为虚析构函数，完成动态绑定
31     delete animal;
32
33     return 0;
34 }
```

虚析构函数也可以做成纯虚析构函数，如果一个类中包含了纯虚析构函数，那么这个类依然是一个抽象类，无法实例化对象。

总结：

如果一个类的目的不是为了实现多态，仅仅是作为一个基类来使用，那么无需将析构函数设置为虚析构函数。

如果一个类的目的就是为了实现多态的，那么这个类的析构函数就有必要设置为虚析构函数。

2.10. 结构体

2.10.1. 结构体的定义与使用

在C++中，还有一种用户自定义的数据类型，结构体。结构体的定义与使用基本与类相同。

```

1 // 定义结构体
2 struct Student {
3     // 结构体中定义的属性
4     string name;
5     int age;
6
7     // 结构体中的构造函数
8     Student() {
9         name = "";
10        age = 0;
11    }
12
13    Student(string name, int age): name(name), age(age) {}
```

```

15 // 结构体中的函数
16 void study() const {
17     cout << name << "正在努力学习" << endl;
18 }
19
20 // 结构体中的析构函数
21 ~Student() {
22     cout << "结构体析构函数" << endl;
23 }
24 };
25
26 int main() {
27     // 创建结构体对象
28     // 创建结构体对象时候的关键字struct可以省略不写
29     struct Student student;
30     // 访问成员
31     student.name = "zhangsan";
32     student.age = 18;
33     student.study();
34
35     // 通过有参构造创建结构体对象
36     struct Student xiaoming("xiaoming", 12);
37     struct Student xiaohei = Student("xiaohei", 11);
38     struct Student xiaobai = {"xiaobai", 10};
39
40     // 在堆上创建结构体对象
41     struct Student* xiaoli = new Student("xiaoli", 11);
42     delete xiaoli;
43
44     return 0;
45 }
```

2.10.2. 结构体与类的区别

C++对结构体进行了很多的拓展，是的C++对结构体用于与类几乎相同的功能：可以设计属性、函数，可以设计构造、析构，甚至可以有继承，可以有多态。现在看来C++的结构体与类的区别，主要是一点：默认的访问权限不同

- 类成员默认的访问权限是private
- 结构体成员默认的访问权限是public

2.11. 模板

2.11.1. 模板的介绍

c++提供了函数模板(function template)。所谓函数模板，实际上是建立一个通用函数，其函数类型和形参类型不具体制定，用一个虚拟的类型来代表。这个通用函数就称为函数模板。凡是函数体相同的函数都可以用这个模板代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现不同函数的功能。

c++提供两种模板机制：**函数模板**和**类模板**

总结：

模板把函数或类要处理的数据类型参数化，表现为参数的多态性，成为类属。

模板用于表达逻辑结构相同，但具体数据元素类型不同的数据对象的通用行为。

2.11.2. 函数模板

2.11.2.1. 函数模板的定义

```

1 // 需求：我想要设计一个函数，实现两个int变量的值的交换
2 void mySwap(int& a, int& b) {
3     int tmp = a;
4     a = b;
5     b = tmp;
6 }
7 // 需求：我想要设计一个函数，实现两个double变量的值的交换
8 void mySwap(double& a, double& b) {
9     double tmp = a;
10    a = b;
11    b = tmp;
12 }
13 // 那么我需要再对两个float类型的变量进行交换，是不是还需要再写一个函数呢？
14 // 需要交换的变量的类型越多，我就越需要写更多的重复的函数
15 // 而且一旦需求变更了，交换的逻辑需要做一些小小的改变。那么每一个函数我都得修改一下，非常的复杂
16 //
17 // 我如果能够设计一个通用的函数，能够把类型当作参数传递到这个函数中，就可以简化很多很多的工作了！
18 // 这就是函数模板！

```

```

1 // 定义函数模板
2 // template: 模板关键字
3 // typename: 定义虚拟类型关键字，也可以使用class
4 // T: 定义的一个虚拟的类型，在这里暂不确定是什么类型，等到调用这个函数的时候就可以确定了
5 template<typename T>
6 void mySwap(T& a, T& b) {
7     T tmp = a;
8     a = b;
9     b = tmp;
10 }

```

2.11.2.2. 函数模板的使用

```

1 template<class T>
2 void mySwap(T& a, T& b) {
3     T tmp = a;
4     a = b;
5     b = tmp;
6 }
7
8 int main() {
9     int a = 10, b = 20;
10    double x = 3.14, y = 0.99;
11
12    // 1. 显式指定类型
13    mySwap<int>(a, b);
14
15    // 2. 可以自动根据实参的类型进行推导
16    mySwap(a, b);           // 这里调用的mySwap中，类型T被推导为int类型
17    mySwap(x, y);          // 这里调用的mySwap中，类型T被推导为double类型
18
19    // 注意事项：类型推导的时候，需要保证一致性。不满足一致性无法推导。
20    // 例如 mySwap(a, y);
21    // 第一个实参a是int类型，推导T的类型为int；第二个实参y是double类型，推导T的类型为double；
22    // 不一致
23
24    return 0;
}

```

2.11.2.3. 函数模板案例

```

1 // 需求：定义一个模板函数，实现对一个数组中对元素进行升序排序
2 template<class T>
3 void mySort(T array[], int len) {
4     for (int i = 0; i < len - 1; i++) {
5         int minIndex = i;
6         for (int j = i + 1; j < len; j++) {
7             if (array[minIndex] > array[j]) {
8                 minIndex = j;
9             }
10        }
11        if (minIndex != i) {
12            T tmp = array[minIndex];
13            array[minIndex] = array[i];
14            array[i] = tmp;
15        }
16    }
17 }

```

```

19 // 需求：定义一个模板函数，实现将一个数组中对元素拼接成为字符串返回
20 template<class T>
21 void showArray(T array[], int len) {
22     if (len == 0) {
23         cout << "[]" << endl;
24         return;
25     }
26     cout << "[";
27     for (int i = 0; i < len - 1; i++) {
28         cout << array[i] << ", ";
29     }
30     cout << array[len - 1] << "]" << endl;
31 }
32
33 int main() {
34
35     // 定义一个int[]
36     int array1[] = {1, 3, 5, 7, 9, 0, 8, 6, 4, 2};
37     int len1 = sizeof(array1) / sizeof(int);
38     mySort(array1, len1);
39     showArray(array1, len1);
40
41     // 定义一个double[]
42     double array2[] = {3.14, 9.28, 3, 3.44, -9.2, 8.22};
43     int len2 = sizeof(array2) / sizeof(double);
44     mySort(array2, len2);
45     showArray(array2, len2);
46
47     // 定义一个char[]
48     char array3[] = {'a', 'l', '1', 'm', 'k'};
49     int len3 = sizeof(array3) / sizeof(char);
50     mySort(array3, len3);
51     showArray(array3, len3);
52
53     return 0;
54 }
```

2.11.2.4. 函数模板与普通函数

函数模板和普通函数在调用的时候，需要注意：

- 普通函数调用，是可以发生自动的类型转换的；函数模板调用，是不可以发生自动的类型转换的
- 如果调用函数的时候，实参既可以匹配普通函数，又可以匹配函数模板，则优先匹配普通函数

```

1 // 定义一个函数模板
2 template<class T>
3 int myPlus(const T& n1, const T& n2) {
4     return n1 + n2;
5 }
```

```

6
7 int myPlus(int n1, int n2) {
8     return n1 + n2;
9 }
10
11 int main() {
12
13     // 调用普通函数，类型可以自动转换
14     int n1 = 10;
15     char c = 'a';
16     myPlus(n1, c);           // 这里进行了类型的自动转换，c是char类型，被转型成了int类型。
17
18     // 调用函数模板，类型不可以自动转换
19     // myPlus(n1, c);          // 这里直接错误，因为这里不允许类型的转换。
20
21     // 如果实参既可以匹配普通函数，又可以匹配函数参数，则优先普通函数调用。
22     myPlus(10, 20);
23
24     return 0;
25 }
```

2.11.2.5. 函数模板的局限性

函数模板虽然很通用，但是并不是万能的，有时候也会有不匹配的情况出现。

```

1 template<class T>
2 bool compare(const T& t1, const T& t2) {
3     return t1 > t2;
4 }
```

对于上述的函数模板来说，如果是比较整型、浮点型甚至字符类型的数据都是没有问题的。可是如果我设置为Person类型呢？两个Person对象无法使用>进行比较，这里自然也就出问题了。

那么如何解决这样的问题呢？

1. 重载运算符，重载>运算符。
2. 通过函数模板的重载来解决。

函数模板的重载，就是为了解决特定类型的对象的问题，通过函数模板的重载，可以为这些特定的数据类型提供具像化的模板。

```

1 class Person {
2 public:
3     int age;
4 };
5
6 template<class T>
7 bool compare(const T& t1, const T& t2) {
8     return t1 > t2;
```

```

9 }
10
11 template<>
12 bool compare<Person>(const Person& p1, const Person& p2) {
13     return p1.age > p2.age;
14 }
15
16
17 int main() {
18
19     Person p1;
20     p1.age = 15;
21
22     Person p2;
23     p2.age = 12;
24
25     cout << compare(p1, p2) << endl;
26
27
28     return 0;
29 }
```

2.11.3. 类模板

2.11.3.1. 类模板的定义

类模板和函数模板的定义和使用基本是一样的，如何定义函数模板，就如何定义类模板。但是类模板与函数模板还是有点区别的：

- 类模板不能自动类型推导。

```

1 template<class T1, class T2 = int>
2 class NumberOperator {
3 public:
4     T1 num1;
5     T2 num2;
6
7     void cal() {
8         cout << num1 + num2 << endl;
9     }
10 };
11
12 int main() {
13
14     // 创建对象，不能类型推导，只能自己指定类型
15     NumberOperator<int, int> op1;
16     op1.num1 = 10;
```

```

17     op1.num2 = 20;
18     op1.cal();
19
20     // 创建对象
21     NumberOperator<double> op2;
22     op2.num1 = 3.14;
23     op2.num2 = 10;
24     op2.cal();
25
26     return 0;
27 }
```

2.11.3.2. 类模板做函数参数

```

1 template<class T1, class T2 = int>
2 class NumberOperation {
3 public:
4     T1 num1;
5     T2 num2;
6
7     void cal() {
8         cout << num1 + num2 << endl;
9     }
10 };
11
12 // 参数中明确模板类
13 void useNumberOperation(NumberOperation<int, int>& op) {
14     op.cal();
15 }
16
17 // 参数中使用模板
18 template<typename T1, typename T2>
19 void useNumberOperation02(NumberOperation<T1, T2>& op) {
20     op.cal();
21 }
22
23 int main() {
24
25     // 参数明确模板类调用
26     NumberOperation<int, int> op;
27     op.num1 = 10;
28     op.num2 = 20;
29     useNumberOperation(op);
30
31     // 参数模板
32     useNumberOperation02(op);
33     NumberOperation<double, int> op2;
34     op2.num1 = 10.5;
```

```

35     op2.num2 = 5;
36     useNumberOperation02(op2);
37
38     return 0;
39 }
```

2.11.3.3. 类模板继承

```

1 // 定义模板类
2 template<typename T>
3 class Animal {
4 public:
5     T arg;
6 };
7
8 // 普通类继承模板类的时候，必须明确指定类型
9 class Dog: Animal<int> {
10    // 这里继承到的arg的数据类型是int
11 };
12
13 template<typename E>
14 class Person: Animal<E> {
15    // 这里继承到的arg的数据类型是E
16 };
```

2.11.3.4. 类模板中的成员函数创建时机

类模板中的成员函数在编译的时候是不会创建的，是在调用这个函数的时候创建。

```

1 class Dog {
2 public:
3     void bark() { cout << "汪汪" << endl; }
4 }
5
6 class Cat {
7 public:
8     void sleep() { cout << "呼呼" << endl; }
9 }
10
11 template<typename T>
12 class Person {
13 public:
14     T pet;
15
16     void makeBark() {
17         pet.bark();
```

```

18     }
19
20     void makeSleep() {
21         pet.sleep();
22     }
23 }
24
25 int main() {
26     // 在类设计完成后，直接编译程序，发现是没有问题的。
27
28     // 调用makeBark函数的时候，也是没有问题的，可以正常调用。
29     Person<Dog> xiaobai;
30     xiaobai.makeBark();
31
32     // 调用makeSleep函数的时候就出问题了，不能调用了！
33     xiaobai.makeSleep();
34
35     // 原因：类模板中的成员函数是在调用的时候才会创建的！
36     // 因为在编译的时候，只是知道有一个对象是obj，但是具体是什么类型，不知道！
37     // 在调用makeBark的时候，创建了这个函数，而我们设置的类型是Dog类型，没有问题，可以正常执行
38     // 在调用makeSleep的时候，创建了这个函数，判断pet的类型是Dog类型，而在Dog类中不存在sleep函
39     // 数，因此就报错了。
40
41     return 0;
}

```

2.11.3.5. 类模板类外实现

```

1 template<typename T, typename M>
2 class NumberCalculator {
3 private:
4     T n1;
5     M n2;
6 public:
7     NumberCalculator() {}
8     NumberCalculator(T n1, M n2);
9
10    void add();
11 };
12 // 构造函数类外实现
13 template<typename T, typename M>
14 NumberCalculator<T, M>::NumberCalculator(T n1, M n2) {
15     this->n1 = n1;
16     this->n2 = n2;
17 }
18 // 普通函数类外实现
19 template<typename T, typename M>
20 NumberCalculator<T, M>::add() {

```

```

21     cout << n1 + n2 << endl;
22 }
```

2.11.3.6. 类模板头文件和原文件分离问题

我们在写程序的时候，很多时候都是需要将类的声明和实现分开来写。将类的声明部分写到.h文件中，将类的实现部分写在.cpp文件中。在使用到这个类的时候，直接包含.h文件即可。但是，如果是一个模板类，这样做是有问题的。

NumberCalculator.h

```

1 #pragma once
2
3 template<typename T, typename M>
4 class NumberCalculator {
5 private:
6     T n1;
7     M n2;
8 public:
9     NumberCalculator() {}
10    NumberCalculator(T n1, M n2);
11
12    void add();
13};
```

NumberCalculator.cpp

```

1 #include <iostream>
2 #include "NumberCalculator.h"
3 using namespace std;
4
5 // 构造函数类外实现
6 template<typename T, typename M>
7 NumberCalculator<T, M>::NumberCalculator(T n1, M n2) {
8     this->n1 = n1;
9     this->n2 = n2;
10}
11// 普通函数类外实现
12 template<typename T, typename M>
13 void NumberCalculator<T, M>::add() {
14     cout << n1 + n2 << endl;
15}
```

```

1 int main() {
2     // 通过无参构造创建对象，没有问题
3     NumberCalculator<int, int> call1;
4     // 通过有参构造创建对象，出问题了
```

```

5     NumberCalculator<int, int> cal2(10, 20);
6
7     // 问题出现原因:
8     // 我们虽然引入了.h文件，但是模板类中的函数是在调用的时候才会创建的，因此在编译阶段也不会管对应的.cpp文件中的实现部分。
9     // 而到了使用到这个函数的时候，发现这个函数已经创建了，但是没有实现。因此就报错了。
10    // 相当于我们只是在.h中声明了函数，但是并没有实现。
11    //
12    // 如何解决问题:
13    // 1. 使用#include引入cpp文件
14    // 2. 将类的声明和实现放到一个文件中
15    //      这个文件我们习惯上会定义为.hpp文件，但是并不是绝对的，只是一个习惯和约定的问题。
16 }

```

2.11.3.7. 类模板遇到友元

```

1 // 全局友元函数类外实现-03: 定义类
2 template<typename T, typename M>
3 class NumberCalculator;
4
5 // 全局友元函数类外实现-02: 在类之前定义
6 template<typename T, typename M>
7 void printNumberCalculator(const NumberCalculator<T, M>& op) {
8     cout << "n1 = " << op.n1 << ", n2 = " << op.n2 << endl;
9 }
10
11 template<typename T, typename M>
12 class NumberCalculator {
13     // 全局友元函数类内实现，无需进行什么处理，直接在这里写实现即可。
14     /*
15         friend void printNumberCalculator(const NumberCalculator<T, M>& op) {
16             cout << "n1 = " << op.n1 << ", n2 = " << op.n2 << endl;
17         }
18     */
19
20     // 全局友元函数类外实现-01: 在函数的后面添加一对尖括号，表示一个模板函数
21     friend void printNumberCalculator<>(const NumberCalculator<T, M>& op);
22 private:
23     T n1;
24     M n2;
25 public:
26     NumberCalculator();
27     NumberCalculator(T n1, M n2);
28 };
29
30 template<typename T, typename M>
31 NumberCalculator<T, M>::NumberCalculator(T n1, M n2) {
32     this->n1 = n1;

```

```

33     this->n2 = n2;
34 }
35
36 template<typename T, typename M>
37 NumberCalculator<T, M>::NumberCalculator() = default;
38
39
40 int main() {
41
42     NumberCalculator<int, int> op(10, 20);
43
44     printNumberCalculator(op);
45
46     return 0;
47 }
```

第三部分 STL标准模板库

3.1. STL概述

长久以来，软件界一直希望建立一种可重复利用的东西，以及一种得以制造出“可重复运用的东西”的方法，让程序员的心血不止于随时间的迁移，人事异动而烟消云散，从函数(functions)，类别(classes)，函数库(function libraries)，类别库(class libraries)、各种组件，从模块化设计，到面向对象(object oriented)，为的就是复用性的提升。

复用性必须建立在某种标准之上。但是在许多环境下，就连软件开发最基本的数据结构(data structures) 和算法(algorithm)都未能有一套标准。大量程序员被迫从事大量重复的工作，竟然是为了完成前人已经完成而自己手上并未拥有的程序代码，这不仅是人力资源的浪费，也是挫折与痛苦的来源。

为了建立数据结构和算法的一套标准，并且降低他们之间的耦合关系，以提升各自的独立性、弹性、交互操作性(相互合作性, interoperability)，诞生了STL。

3.1.1. STL基本概念

STL(Standard Template Library, 标准模板库)，是惠普实验室开发的一系列软件的统称。现在主要出现在c++中，但是在引入c++之前该技术已经存在很长时间了。

STL从广义上分为：容器(container) 算法(algorithm) 迭代器(iterator)，容器和算法之间通过迭代器进行无缝连接。STL几乎所有的代码都采用了模板类或者模板函数，这相比传统的由函数和类组成的库来说提供了更好的代码重用机会。STL(Standard Template Library)标准模板库，在我们c++标准程序库中隶属于STL的占到了80%以上。

3.1.2. STL六大组件简介

STL提供了六大组件，彼此之间可以组合套用，这六大组件分别是：容器、算法、迭代器、仿函数、适配器（配接器）、空间配置器。

- 容器：**各种数据结构，如vector、list、deque、set、map等，用来存放数据，从实现角度来看，STL容器是一种class template。
- 算法：**各种常用的算法，如sort、find、copy、for_each。从实现的角度来看，STL算法是一种

```
function tempalte.
```

- **迭代器**: 扮演了容器与算法之间的胶合剂, 共有五种类型, 从实现角度来看, 迭代器是一种将operator*, operator->, operator++, operator--等指针相关操作予以重载的class template. 所有STL容器都附带有自己专属的迭代器, 只有容器的设计者才知道如何遍历自己的元素。原生指针(native pointer)也是一种迭代器。
- **仿函数**: 行为类似函数, 可作为算法的某种策略。从实现角度来看, 仿函数是一种重载了operator()的class 或者class template
- **适配器**: 一种用来修饰容器或者仿函数或迭代器接口的东西。
- **空间配置器**: 负责空间的配置与管理。从实现角度看, 配置器是一个实现了动态空间配置、空间管理、空间释放的class tempalte.

STL六大组件的交互关系, 容器通过空间配置器取得数据存储空间, 算法通过迭代器存储容器中的内容, 仿函数可以协助算法完成不同的策略的变化, 适配器可以修饰仿函数。

3.1.3. STL优点

STL 是 C++的一部分, 因此不用额外安装什么, 它被内建在你的编译器之内。

STL 的一个重要特性是将数据和操作分离。数据由容器类别加以管理, 操作则由可定制的算法定义。迭代器在两者之间充当“粘合剂”, 以使算法可以和容器交互运作。程序员可以不用思考 STL 具体的实现过程, 只要能够熟练使用 STL 就 OK 了。这样他们就可以把精力放在程序开发的别的方面。

STL 具有高可重用性, 高性能, 高移植性, 跨平台的优点。

- **高可重用性**: STL 中几乎所有的代码都采用了模板类和模版函数的方式实现, 这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。关于模板的知识, 已经给大家介绍了。
- **高性能**: 如 map 可以高效地从十万条记录里面查找出指定的记录, 因为 map 是采用红黑树的变体实现的。
- **高移植性**: 如在项目 A 上用 STL 编写的模块, 可以直接移植到项目 B 上。

3.2. STL三大组件

3.2.1. 容器

容器, 置物之所也。

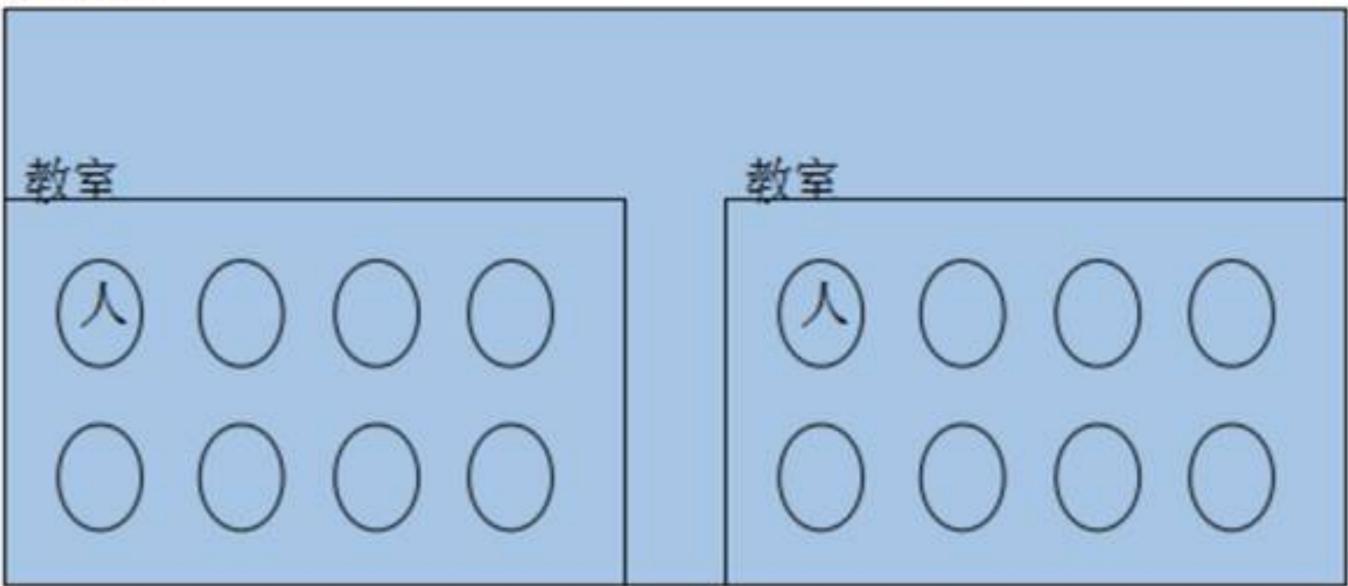
研究数据的特定排列方式, 以利于搜索或排序或其他特殊目的, 这一门学科我们称为数据结构。大学信息类相关专业里面, 与编程最有直接关系的学科, 首推数据结构与算法。几乎可以说, 任何特定的数据结构都是为了实现某种特定的算法。STL容器就是将运用最广泛的一些数据结构实现出来。

常用的数据结构: 数组(array), 链表(list), tree(树), 栈(stack), 队列(queue), 集合(set), 映射表(map), 根据数据在容器中的排列特性, 这些数据分为序列式容器和关联式容器两种。

- 序列式容器强调值的排序, 序列式容器中的每个元素均有固定的位置, 除非用删除或插入的操作改变这个位置。Vector容器、Deque容器、List容器等。
- 关联式容器是非线性的树结构, 更准确的说是二叉树结构。各元素之间没有严格的物理上的顺序关系, 也就是说元素在容器中并没有保存元素置入容器时的逻辑顺序。关联式容器另一个显著特点是: 在值中选择一个值作为关键字key, 这个关键字对值起到索引的作用, 方便查找。Set/multiset容器 Map/multimap容器

容器可以嵌套容器!

教学楼



3.2.2. 算法

算法，问题之解法也。

以有限的步骤，解决逻辑或数学上的问题，这一门学科我们叫做算法(Algorithms).

广义而言，我们所编写的每个程序都是一个算法，其中的每个函数也是一个算法，毕竟它们都是用来解决或大或小的逻辑问题或数学问题。STL收录的算法经过了数学上的效能分析与证明，是极具复用价值的，包括常用的排序，查找等等。特定的算法往往搭配特定的数据结构，算法与数据结构相辅相成。

算法分为：**质变算法** 和 **非质变算法**。

- 质变算法：是指运算过程中会更改区间内的元素的内容。例如拷贝，替换，删除等等
- 非质变算法：是指运算过程中不会更改区间内的元素内容，例如查找、计数、遍历、寻找极值等等

3.2.3. 迭代器

迭代器(iterator)是一种抽象的设计概念，现实程序语言中并没有直接对应于这个概念的实物。iterator定义如下：提供一种方法，使之能够依序寻访某个容器所含的各个元素，而又无需暴露该容器的内部表示方式。

迭代器的设计思维-STL的关键所在，STL的中心思想在于将容器(container)和算法(algorithms)分开，彼此独立设计，最后再一贴胶着剂将他们撮合在一起。从技术角度来看，容器和算法的泛型化并不困难，c++的class template和function template可分别达到目标，如何设计出两这个之间的良好的胶着剂，才是大难题。

迭代器的种类：

输入迭代器	提供对数据的只读访问	只读，支持++、==、!=
输出迭代器	提供对数据的只写访问	只写，支持++
前向迭代器	提供读写操作，并能向前推进迭代器	读写，支持++、==、!=
双向迭代器	提供读写操作，并能向前和向后操作	读写，支持++、--，
随机访问迭代器	提供读写操作，并能以跳跃的方式访问容器的任意数据，是功能最强的迭代器	读写，支持++、--、[n]、-n、<、<=、>、>=

3.3. 常用容器

3.3.1. string容器

3.3.1.1. string容器基本概念

C风格字符串(以空字符结尾的字符数组)太过复杂难于掌握，不适合大程序的开发，所以C++标准库定义了一种string类。

C++的字符串与C语言的字符串比较

- C语言: char* 是一个指针。
- C++:
 - string是一个类，内部封装了char*，用来管理这个容器。
 - string类中封装了很多的功能函数，非常实用。例如: find、copy、delete、replace、insert 等。
 - 不用考虑内存释放和越界的问题。

string管理char*所分配的内存。每一次string的复制，取值都由string类负责维护，不用担心复制越界和取值越界等。

3.3.1.2. string容器常用操作

1. string构造函数

```

1 string(); //创建一个空的字符串 例如: string str;
2 string(const string& str); //使用一个string对象初始化另一个string对象
3 string(const char* s); //使用字符串s初始化
4 string(int n, char c); //使用n个字符c初始化

```

2. string基本赋值操作

```

1 string& operator=(const char* s); //char*类型字符串 赋值给当前的字符串
2 string& operator=(const string &s); //把字符串s赋给当前的字符串
3 string& operator=(char c); //字符赋值给当前的字符串
4 string& assign(const char *s); //把字符串s赋给当前的字符串
5 string& assign(const char *s, int n); //把字符串s的前n个字符赋给当前的字符串
6 string& assign(const string &s); //把字符串s赋给当前字符串
7 string& assign(int n, char c); //用n个字符c赋给当前字符串
8 string& assign(const string &s, int start, int n); //将s从start开始n个字符赋值给字符串

```

3. string存取字符操作

```

1 char& operator[](int n); //通过[]方式取字符
2 char& at(int n); //通过at方法获取字符

```

4. string拼接操作

```

1 string& operator+=(const string& str); //重载+=操作符
2 string& operator+=(const char* str); //重载+=操作符
3 string& operator+=(const char c); //重载+=操作符
4 string& append(const char *s); //把字符串s连接到当前字符串结尾
5 string& append(const char *s, int n); //把字符串s的前n个字符连接到当前字符串结尾
6 string& append(const string &s); //同operator+=( )
7 string& append(const string &s, int pos, int n); //把字符串s中从pos开始的n个字符连接到当前字符串结尾
8 string& append(int n, char c); //在当前字符串结尾添加n个字符c

```

5. string查找和替换

```

1 int find(const string& str, int pos = 0) const; //查找str第一次出现位置,从pos开始查找
2 int find(const char* s, int pos = 0) const; //查找s第一次出现位置,从pos开始查找
3 int find(const char* s, int pos, int n) const; //从pos位置查找s的前n个字符第一次位置
4 int find(const char c, int pos = 0) const; //查找字符c第一次出现位置
5 int rfind(const string& str, int pos = npos) const; //查找str最后一次位置,从pos开始查找
6 int rfind(const char* s, int pos = npos) const; //查找s最后一次出现位置,从pos开始查找
7 int rfind(const char* s, int pos, int n) const; //从pos查找s的前n个字符最后一次位置
8 int rfind(const char c, int pos = 0) const; //查找字符c最后一次出现位置
9 string& replace(int pos, int n, const string& str); //替换从pos开始n个字符为字符串str
10 string& replace(int pos, int n, const char* s); //替换从pos开始的n个字符为字符串s

```

6. string比较操作

```

1  /*
2  compare函数在>时返回 1, <时返回 -1, ==时返回 0。
3  比较区分大小写，比较时参考字典顺序，排越前面的越小。
4  大写的A比小写的a小。
5  */
6  int compare(const string &s) const;//与字符串s比较
7  int compare(const char *s) const;//与字符串s比较

```

7. string子串

```
1 | string substr(int pos = 0, int n = npos) const; //返回由pos开始的n个字符组成的字符串
```

8. string插入和删除操作

```

1 | string& insert(int pos, const char* s); //插入字符串
2 | string& insert(int pos, const string& str); //插入字符串
3 | string& insert(int pos, int n, char c); //在指定位置插入n个字符c
4 | string& erase(int pos, int n = npos); //删除从Pos开始的n个字符

```

9. string和C语言风格字符串转换

```

1 //string 转 char*
2 string str = "itcast";
3 const char* cstr = str.c_str();
4 //char* 转 string
5 char* s = "itcast";
6 string str(s);

```

1 在c++中存在一个从const char到string的隐式类型转换，却不存在从一个string对象到CString的自动类型转换。对于string类型的字符串，可以通过cstr()函数返回string对象对应的C_string。
2 通常，程序员在整个程序中应坚持使用string类对象，直到必须将内容转化为char时才将其转换为C_string。

提示：

为了修改string字符串的内容，下标操作符[]和at都会返回字符的引用。但当字符串的内存被重新分配之后，可能发生错误。

```
1 | string s = "abcdefg";
```

```

2 char& a = s[2];
3 char& b = s[3];
4
5 a = '1';
6 b = '2';
7
8 cout << s << endl;
9 cout << (int*)s.c_str() << endl;
10
11 s = "pppppppppppppppppppppppp";
12
13 //a = '1';
14 //b = '2';
15
16 cout << s << endl;
17 cout << (int*)s.c_str() << endl;

```

3.3.2. vector容器

3.3.2.1. vector容器基本概念

vector的数据安排以及操作方式，与array非常相似，两者的唯一差别在于空间的运用的灵活性。Array是静态空间，一旦配置了就不能改变，要换大一点或者小一点的空间，可以，一切琐碎得由自己来，首先配置一块新的空间，然后将旧空间的数据搬往新空间，再释放原来的空间。Vector是动态空间，随着元素的加入，它的内部机制会自动扩充空间以容纳新元素。因此vector的运用对于内存的合理利用与运用的灵活性有很大的帮助，我们再也不必害怕空间不足而一开始就要求一个大块头的array了。

3.3.2.2. vector容器常用操作

1. vector的构造函数

```

1 vector<T> v; //采用模板实现类实现，默认构造函数
2 vector(v.begin(), v.end()); //将v[begin(), end()]区间中的元素拷贝给本身。
3 vector(n, elem); //构造函数将n个elem拷贝给本身。
4 vector(const vector &vec); //拷贝构造函数。
5
6 //例子 使用第二个构造函数 我们可以...
7 int arr[] = {2,3,4,1,9};
8 vector<int> v1(arr, arr + sizeof(arr) / sizeof(int));

```

2. vector的常用赋值函数

```

1 assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
2 assign(n, elem); //将n个elem拷贝赋值给本身。
3 vector& operator=(const vector &vec); //重载等号操作符
4 swap(vec); // 将vec与本身的元素互换。

```

3. vector的大小操作

```

1 size(); //返回容器中元素的个数
2 empty(); //判断容器是否为空
3 resize(int num); //重新指定容器的长度为num, 若容器变长, 则以默认值填充新位置。如果容器变短, 则末尾超出容器长度的元素被删除。
4 resize(int num, elem); //重新指定容器的长度为num, 若容器变长, 则以elem值填充新位置。如果容器变短, 则末尾超出容器长>度的元素被删除。
5 capacity(); //容器的容量
6 reserve(int len); //容器预留len个元素长度, 预留位置不初始化, 元素不可访问。

```

4. vector的数据存取操作

```

1 at(int idx); //返回索引idx所指的数据, 如果idx越界, 抛出out_of_range异常。
2 operator[]; //返回索引idx所指的数据, 越界时, 运行直接报错
3 front(); //返回容器中第一个数据元素
4 back(); //返回容器中最后一个数据元素

```

5. vector插入和删除操作

```

1 insert(const_iterator pos, int count, ele); //迭代器指向位置pos插入count个元素ele。
2 push_back(ele); //尾部插入元素ele
3 pop_back(); //删除最后一个元素
4 erase(const_iterator start, const_iterator end); //删除迭代器从start到end之间的元素
5 erase(const_iterator pos); //删除迭代器指向的元素
6 clear(); //删除容器中所有元素

```

3.3.2.3. vector迭代器

Vector维护一个线性空间, 所以不论元素的型别如何, 普通指针都可以作为vector的迭代器, 因为vector迭代器所需要的操作行为, 如operator, operator->, operator++, operator--, operator+, operator-, operator+=, operator-=, 普通指针天生具备。Vector支持随机存取, 而普通指针正有着这样的能力。所以vector提供的是随机访问迭代器(Random Access Iterators)。

```

1 // vector提供了begin()函数, 用来返回指向首元素的指针
2 // vector提供了end()函数, 用来返回指向最后一位元素的下一位的指针
3

```

```

4 // 创建容器
5 vector<int> v;
6 // 向容器中添加元素
7 v.push_back(1);
8 v.push_back(3);
9 v.push_back(5);
10 v.push_back(7);
11 v.push_back(9);
12 // 使用迭代器遍历容器
13 for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
14     cout << *it << endl;
15 }
16
17 // 使用迭代器倒序遍历容器
18 for (vector<int>::iterator it = v.end(); it != v.begin(); ) {
19     it--;
20     cout << *it << endl;
21 }
22
23 // 迭代器遍历容器，可以缩写为for循环的写法
24 for (int& it : v) {
25     cout << it << endl;
26 }
27
28 // 在使用迭代器遍历容器的过程中，可以通过指针或者是引用来修改到容器中的值

```

3.3.2.4. vector小案例

1. 巧用swap收缩内存空间

```

1 #include<iostream>
2 #include<vector>
3 using namespace std;
4
5 int main(){
6
7     vector<int> v;
8     for (int i = 0; i < 100000;i ++){
9         v.push_back(i);
10    }
11
12    cout << "capacity:" << v.capacity() << endl;
13    cout << "size:" << v.size() << endl;
14
15    //此时 通过resize改变容器大小
16    v.resize(10);
17
18    cout << "capacity:" << v.capacity() << endl;

```

```

19     cout << "size:" << v.size() << endl;
20
21 //容量没有改变
22 vector<int>(v).swap(v);
23
24 cout << "capacity:" << v.capacity() << endl;
25 cout << "size:" << v.size() << endl;
26
27
28 system("pause");
29 return EXIT_SUCCESS;
30 }
```

2. reserve预留空间

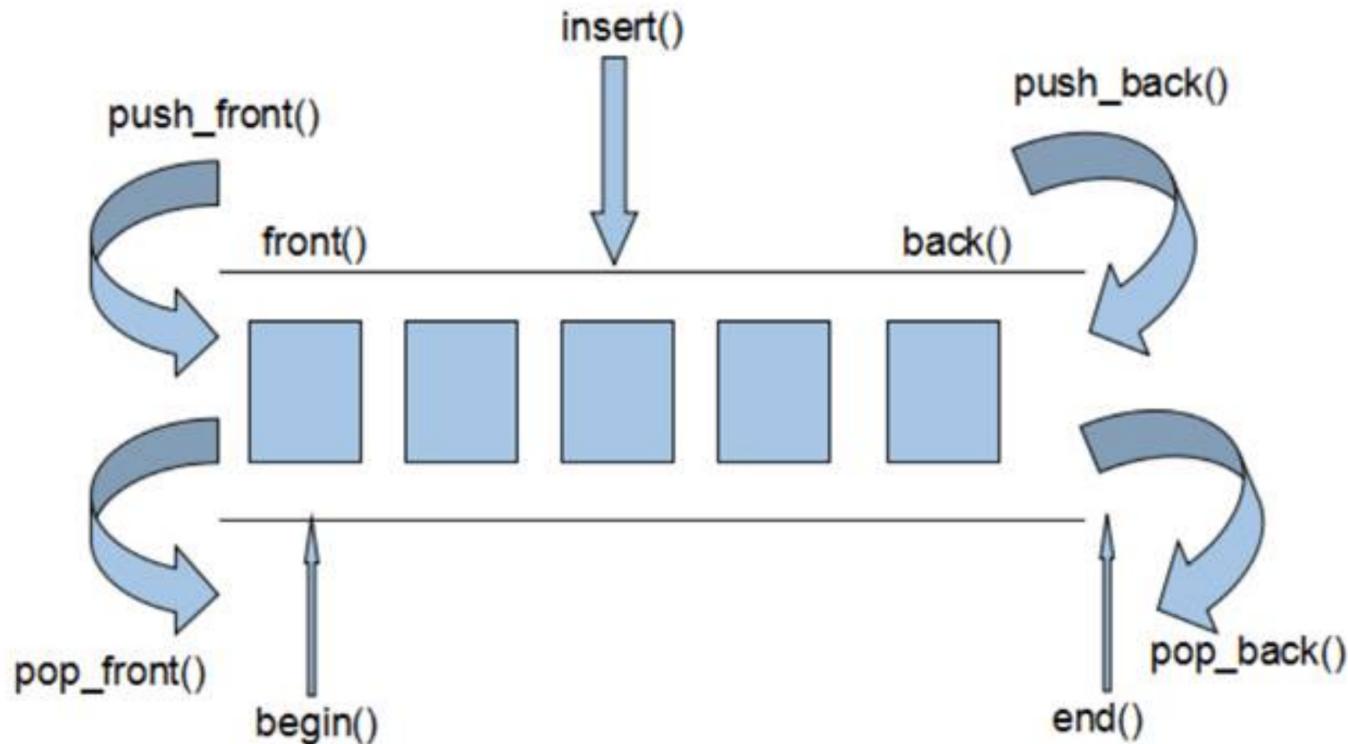
```

1 #include<iostream>
2 #include<vector>
3 using namespace std;
4
5 int main(){
6
7     vector<int> v;
8
9     //预先开辟空间
10    v.reserve(100000);
11
12    int* pStart = NULL;
13    int count = 0;
14    for (int i = 0; i < 100000;i ++){
15        v.push_back(i);
16        if (pStart != &v[0]){
17            pStart = &v[0];
18            count++;
19        }
20    }
21
22    cout << "count:" << count << endl;
23
24    system("pause");
25    return EXIT_SUCCESS;
26 }
```

3.3.3. deque容器

3.3.3.1. deque容器基本概念

Vector容器是单向开口的连续内存空间，deque则是一种双向开口的连续线性空间。所谓的双向开口，意思是可以在头尾两端分别做元素的插入和删除操作，当然，vector容器也可以在头尾两端插入元素，但是在其头部操作效率奇差，无法被接受。



deque容器和vector容器最大的差异，一在于deque允许对头端进行元素的插入和删除操作。二在于deque没有容量的概念，因为它是动态的以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，换句话说，像vector那样，“旧空间不足而重新配置一块更大空间，然后复制元素，再释放旧空间”这样的事情在deque身上是不会发生的。也因此，deque没有必要提供所谓的空间保留(reserve)功能。

虽然deque容器也提供了Random Access Iterator，但是它的迭代器并不是普通的指针，其复杂度和vector不是一个量级，这当然影响各个运算的层面。因此，除非有必要，我们应该尽可能的使用vector，而不是deque。对deque进行的排序操作，为了最高效率，可将deque先完整的复制到一个vector中，对vector容器进行排序，再复制回deque。

3.3.3.2. deque容器常用操作

1. deque构造函数

```

1  deque<T> deqT; //默认构造形式
2  deque(beg, end); //构造函数将[beg, end)区间中的元素拷贝给本身。
3  deque(n, elem); //构造函数将n个elem拷贝给本身。
4  deque(const deque &deq); //拷贝构造函数。

```

2. deque赋值操作

```

1 assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
2 assign(n, elem); //将n个elem拷贝赋值给本身。
3 deque& operator=(const deque &deq); //重载等号操作符
4 swap(deq); // 将deq与本身的元素互换

```

3. deque大小操作

```

1 deque.size(); //返回容器中元素的个数
2 deque.empty(); //判断容器是否为空
3 deque.resize(num); //重新指定容器的长度为num, 若容器变长, 则以默认值填充新位置。如果容器变短,
则末尾超出容器长度的元素被删除。
4 deque.resize(num, elem); //重新指定容器的长度为num, 若容器变长, 则以elem值填充新位置, 如果
容器变短, 则末尾超出容器长度的元素被删除。

```

4. deque双端操作和删除

```

1 push_back(elem); //在容器尾部添加一个数据
2 push_front(elem); //在容器头部插入一个数据
3 pop_back(); //删除容器最后一个数据
4 pop_front(); //删除容器第一个数据

```

5. deque数据存取

```

1 at(idx); //返回索引idx所指的数据, 如果idx越界, 抛出out_of_range。
2 operator[](); //返回索引idx所指的数据, 如果idx越界, 不抛出异常, 直接出错。
3 front(); //返回第一个数据。
4 back(); //返回最后一个数据

```

6. deque插入操作

```

1 insert(pos, elem); //在pos位置插入一个elem元素的拷贝, 返回新数据的位置。
2 insert(pos, n, elem); //在pos位置插入n个elem数据, 无返回值。
3 insert(pos, beg, end); //在pos位置插入[beg, end)区间的数据, 无返回值。

```

7. deque删除操作

```

1 | clear(); // 移除容器的所有数据
2 | erase(beg, end); // 删除[beg, end)区间的数据，返回下一个数据的位置。
3 | erase(pos); // 删除pos位置的数据，返回下一个数据的位置。

```

3.3.3.3. deque小案例

有5名选手：选手ABCDE，10个评委分别对每一名选手打分，去除最高分，去除评委中最低分，取平均分。

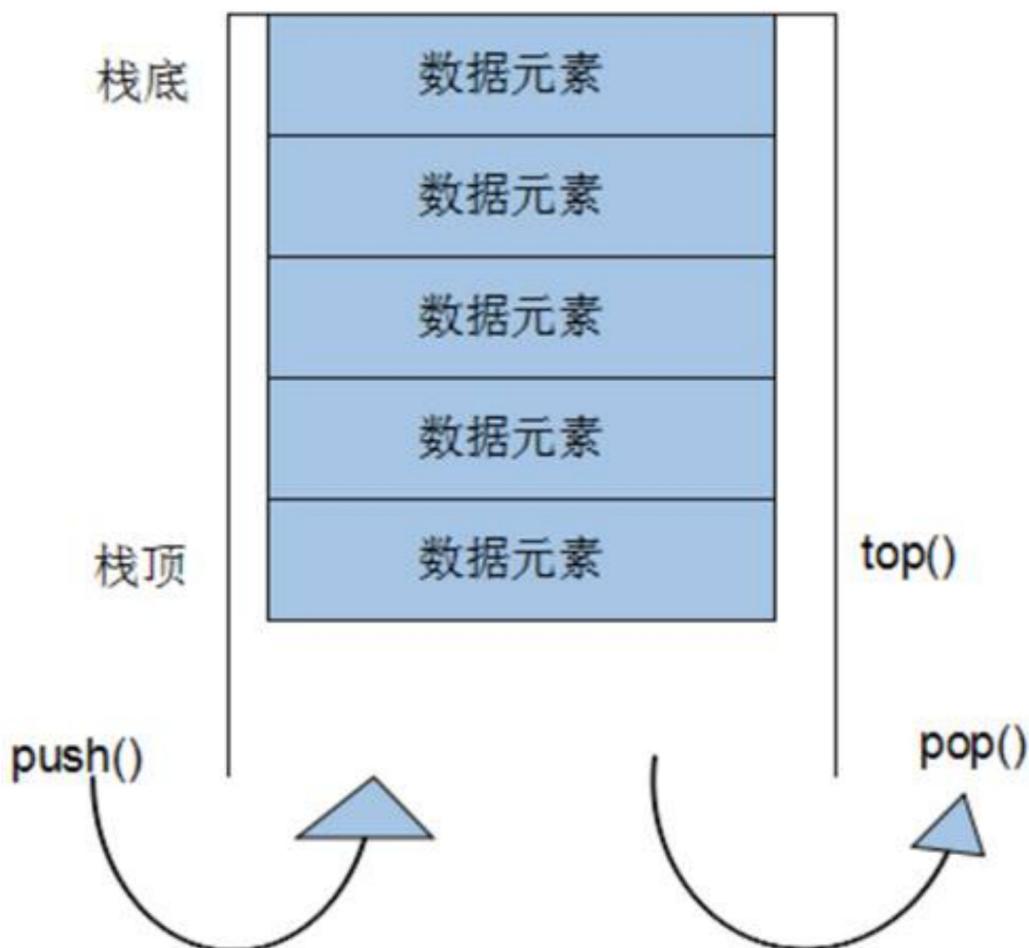
1. 创建五名选手，放到vector中
2. 遍历vector容器，取出来每一个选手，执行for循环，可以把10个评分打分存到deque容器中
3. sort算法对deque容器中分数排序，pop_back pop_front去除最高和最低分
4. deque容器遍历一遍，累加分数，累加分数/d.size()
5. person.score = 平均分

3.3.4. stack容器

3.3.4.1. stack容器基本概念

stack是一种先进后出(First In Last Out, FILO)的数据结构，它只有一个出口，形式如图所示。stack容器允许新增元素，移除元素，取得栈顶元素，但是除了最顶端外，没有任何其他方法可以存取stack的其他元素。换言之，stack不允许有遍历行为。

有元素推入栈的操作称为：push，将元素推出stack的操作称为pop。



stack是没有迭代器的：

Stack所有元素的进出都必须符合“先进后出”的条件，只有stack顶端的元素，才有机会被外界取用。Stack不提供遍历功能，也不提供迭代器。

3.3.4.2. stack容器常用操作

1. 构造函数

```
1 | stack<T> stkT; //stack采用模板类实现， stack对象的默认构造形式：  
2 | stack(const stack &stk); //拷贝构造函数
```

2. 赋值操作

```
1 | stack& operator=(const stack &stk); //重载等号操作符
```

3. 数据存取操作

```
1 | push(elem); //向栈顶添加元素  
2 | pop(); //从栈顶移除第一个元素  
3 | top(); //返回栈顶元素
```

4. 大小操作

```
1 | empty(); //判断堆栈是否为空  
2 | size(); //返回堆栈的大小
```

3.3.5. queue容器

3.3.5.1. queue容器基本概念

Queue是一种先进先出(First In First Out,FIFO)的数据结构，它有两个出口，queue容器允许从一端新增元素，从另一端移除元素。



queue容器没有迭代器：Queue所有元素的进出都必须符合“先进先出”的条件，只有queue的顶端元素，才有机会被外界取用。Queue不提供遍历功能，也不提供迭代器。

3.3.5.2. queue容器常用操作

1. 构造函数

```
1 | queue<T> queT; //queue采用模板类实现, queue对象的默认构造形式:  
2 | queue(const queue &que); //拷贝构造函数
```

2. 存取、插入、删除操作

```
1 | push(elem); //往队尾添加元素  
2 | pop(); //从队头移除第一个元素  
3 | back(); //返回最后一个元素  
4 | front(); //返回第一个元素
```

3. 赋值操作

```
1 | queue& operator=(const queue &que); //重载等号操作符
```

4. 大小操作

```
1 | empty(); //判断队列是否为空  
2 | size(); //返回队列的大小
```

3.3.6. list容器

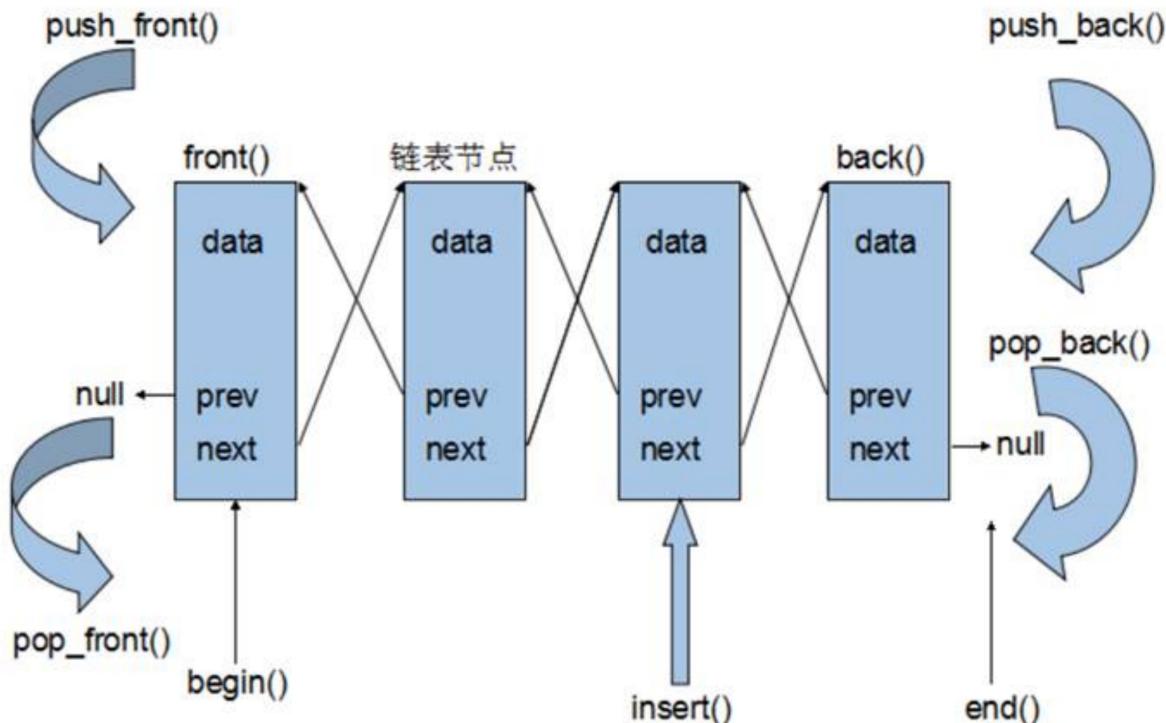
3.3.6.1. list容器基本概念

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

相较于vector的连续线性空间，list就显得复杂许多，它的好处是每次插入或者删除一个元素，就是配置或者释放一个元素的空间。因此，list对于空间的运用有绝对的精准，一点也不浪费。而且，对于任何位置的元素插入或元素的移除，list永远是常数时间。

List和vector是两个最常被使用的容器。

List容器是一个双向链表。



采用动态存储分配，不会造成内存浪费和溢出

链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素

链表灵活，但是空间和时间额外耗费较大

3.3.6.2. list的迭代器

List容器不能像vector一样以普通指针作为迭代器，因为其节点不能保证在同一块连续的内存空间上。List迭代器必须有能力指向list的节点，并有能力进行正确的递增、递减、取值、成员存取操作。所谓“list正确的递增，递减、取值、成员取用”是指，递增时指向下一个节点，递减时指向下一个节点，取值时取的是节点的数据值，成员取用时取的是节点的成员。

由于list是一个双向链表，迭代器必须能够具备前移、后移的能力，所以list容器提供的是Bidirectional Iterators。

List有一个重要的性质，插入操作和删除操作都不会造成原有list迭代器的失效。这在vector是不成立的，因为vector的插入操作可能造成记忆体重新配置，导致原有的迭代器全部失效，甚至List元素的删除，也只有被删除的那个元素的迭代器失效，其他迭代器不受任何影响。

3.3.6.3. list容器常用操作

1. 构造函数

```

1 list<T> lstT;//list采用模板类实现,对象的默认构造形式:
2 list(beg,end); //构造函数将[beg, end)区间中的元素拷贝给本身。
3 list(n,elem); //构造函数将n个elem拷贝给本身。
4 list(const list &lst); //拷贝构造函数。

```

2. 元素插入和删除操作

```

1 push_back(elem); //在容器尾部加入一个元素
2 pop_back(); //删除容器中最后一个元素
3 push_front(elem); //在容器开头插入一个元素
4 pop_front(); //从容器开头移除第一个元素
5 insert(pos, elem); //在pos位置插elem元素的拷贝, 返回新数据的位置。
6 insert(pos, n, elem); //在pos位置插入n个elem数据, 无返回值。
7 insert(pos, beg, end); //在pos位置插入[beg, end)区间的数据, 无返回值。
8 clear(); //移除容器的所有数据
9 erase(beg, end); //删除[beg, end)区间的数据, 返回下一个数据的位置。
10 erase(pos); //删除pos位置的数据, 返回下一个数据的位置。
11 remove(elem); //删除容器中所有与elem值匹配的元素。

```

3. 大小操作

```

1 size(); //返回容器中元素的个数
2 empty(); //判断容器是否为空
3 resize(num); //重新指定容器的长度为num,
4 // 若容器变长, 则以默认值填充新位置。
5 // 如果容器变短, 则末尾超出容器长度的元素被删除。
6
7 resize(num, elem); //重新指定容器的长度为num,
8 // 若容器变长, 则以elem值填充新位置。
9 // 如果容器变短, 则末尾超出容器长度的元素被删除。

```

4. 赋值操作

```

1 assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
2 assign(n, elem); //将n个elem拷贝赋值给本身。
3 list& operator=(const list &lst); //重载等号操作符
4 swap(lst); //将lst与本身的元素互换。

```

5. 数据存取操作

```

1 front(); //返回第一个元素。
2 back(); //返回最后一个元素

```

6. 反转、排序

```

1 reverse(); //反转链表，比如lst包含1, 3, 5元素，运行此方法后，lst就包含5, 3, 1元素。
2 sort(); //list排序

```

3.3.7. set/multiset容器

3.3.7.1. set/multiset容器基本概念

Set的特性是。所有元素都会根据元素的值自动被排序。Set不允许两个元素有相同的值。

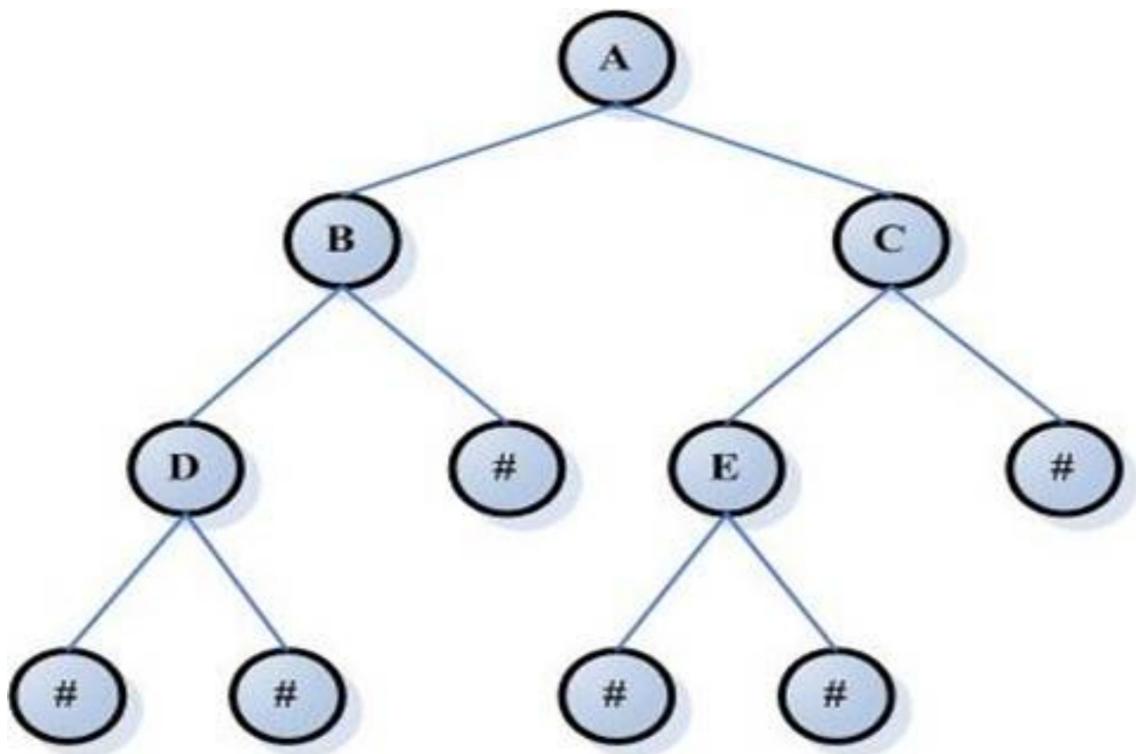
我们可以通过set的迭代器改变set元素的值吗？不行，因为set元素值就是其值，关系到set元素的排序规则。如果任意改变set元素值，会严重破坏set组织。换句话说，set的iterator是一种const_iterator.

set拥有和list某些相同的性质，当对容器中的元素进行插入操作或者删除操作的时候，操作之前所有的迭代器，在操作完成之后依然有效，被删除的那个元素的迭代器必然是一个例外。

multiset特性及用法和set完全相同，唯一的差别在于它允许值重复。set和multiset的底层实现是红黑树，红黑树为平衡二叉树的一种。

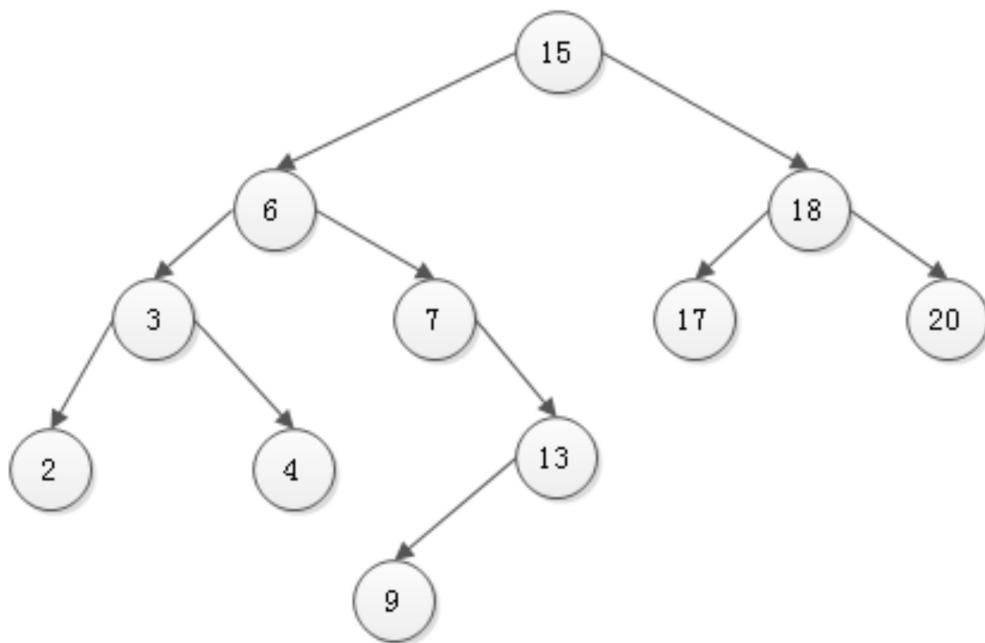
树的简单知识：

二叉树就是任何节点最多只允许有两个字节点。分别是左子结点和右子节点



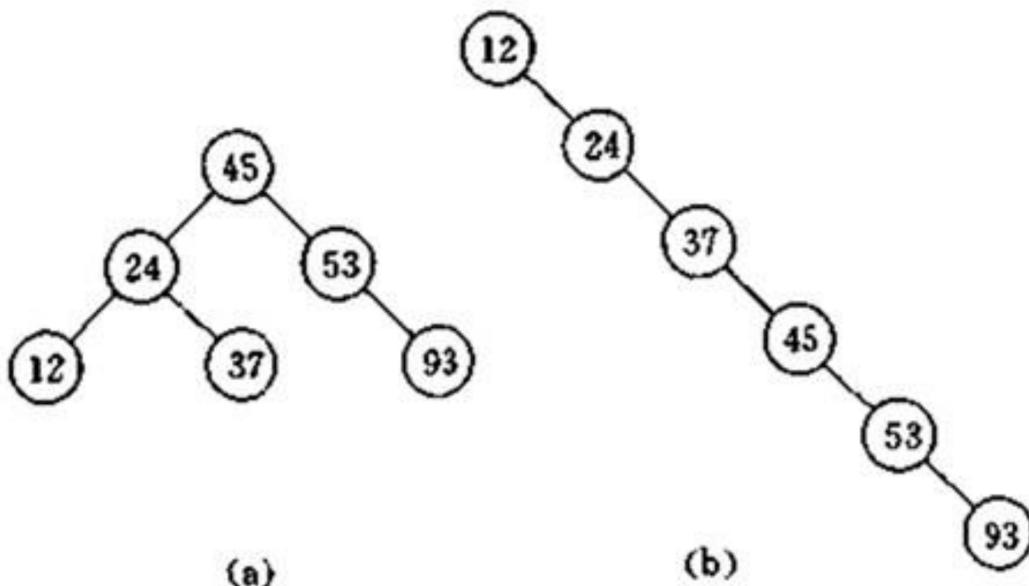
二叉树示意图

二叉搜索树，是指二叉树中的节点按照一定的规则进行排序，使得对二叉树中元素访问更加高效。二叉搜索树的放置规则是：任何节点的元素值一定大于其左子树中的每一个节点的元素值，并且小于其右子树的值。因此从根节点一直向左走，一直到无路可走，即得到最小值，一直向右走，直至无路可走，可得到最大值。那么在二叉搜索树中找到最大元素和最小元素是非常简单的事情。下图为二叉搜索树：



上面我们介绍了二叉搜索树，那么当一个二叉搜索树的左子树和右子树不平衡的时候，那么搜索依据上图表示，搜索9所花费的时间要比搜索17所花费的时间要多，由于我们的输入或者经过我们插入或者删除操作，二叉树失去平衡，造成搜索效率降低。

所以我们有了一个平衡二叉树的概念，所谓的平衡不是指的完全平衡。



3.3.7.2. set/multiset容器常用操作

1. 构造函数

```
1 set<T> st; //set默认构造函数:  
2 multiset<T> mst; //multiset默认构造函数:  
3 set(const set &st); //拷贝构造函数
```

2. set赋值

```
1 set& operator=(const set &st); //重载等号操作符  
2 swap(st); //交换两个集合容器
```

3. set大小操作

```
1 size(); //返回容器中元素的数目  
2 empty(); //判断容器是否为空
```

4. 插入和删除操作

```
1 insert(elem); //在容器中插入元素。  
2 clear(); //清除所有元素  
3 erase(pos); //删除pos迭代器所指的元素，返回下一个元素的迭代器。  
4 erase(beg, end); //删除区间[beg, end)的所有元素，返回下一个元素的迭代器。  
5 erase(elem); //删除容器中值为elem的元素。
```

5. 查找操作

```

1 find(key); //查找键key是否存在,若存在, 返回该键的元素的迭代器; 若不存在, 返回set.end();
2 count(key); //查找键key的元素个数
3 lower_bound(keyElem); //返回第一个key>=keyElem元素的迭代器。
4 upper_bound(keyElem); //返回第一个key>keyElem元素的迭代器。
5 equal_range(keyElem); //返回容器中key与keyElem相等的上下限的两个迭代器。

```

3.3.7.3. 对组

对组(pair)将一对值组合成一个值, 这一对值可以具有不同的数据类型, 两个值可以分别用pair的两个公有属性first和second访问。

类模板: `template <class T1, class T2> struct pair.`

如何创建对组?

```

1 //第一种方法创建一个对组
2 pair<string, int> pair1(string("name"), 20);
3 cout << pair1.first << endl; //访问pair第一个值
4 cout << pair1.second << endl; //访问pair第二个值
5 //第二种
6 pair<string, int> pair2 = make_pair("name", 30);
7 cout << pair2.first << endl;
8 cout << pair2.second << endl;
9 //pair=赋值
10 pair<string, int> pair3 = pair2;
11 cout << pair3.first << endl;
12 cout << pair3.second << endl;

```

3.3.8. map/multimap容器

3.3.8.1. map/multimap基本概念

Map的特性是, 所有元素都会根据元素的键值自动排序。Map所有的元素都是pair, 同时拥有实值和键值, pair的第一元素被视为键值, 第二元素被视为实值, map不允许两个元素有相同的键值。

我们可以通过map的迭代器改变map的键值吗? 答案是不行, 因为map的键值关系到map元素的排列规则, 任意改变map键值将会严重破坏map组织。如果想要修改元素的实值, 那么是可以的。

Map和list拥有相同的某些性质, 当对它的容器元素进行新增操作或者删除操作时, 操作之前的所有迭代器, 在操作完成之后依然有效, 当然被删除的那个元素的迭代器必然是个例外。

Multimap和map的操作类似, 唯一区别multimap键值可重复。

Map和multimap都是以红黑树为底层实现机制。

3.3.8.2. map/multimap常用操作

1. 构造函数

```
1 | map<T1, T2> mapTT; //map默认构造函数:  
2 | map(const map &mp); //拷贝构造函数
```

2. 赋值操作

```
1 | map& operator=(const map &mp); //重载等号操作符  
2 | swap(mp); //交换两个集合容器
```

3. 大小操作

```
1 | size(); //返回容器中元素的数目  
2 | empty(); //判断容器是否为空
```

4. 插入操作

```
1 | map.insert(...); //往容器插入元素，返回pair<iterator,bool>  
2 | map<int, string> mapStu;  
3 | // 第一种 通过pair的方式插入对象  
4 | mapStu.insert(pair<int, string>(3, "小张"));  
5 | // 第二种 通过pair的方式插入对象  
6 | mapStu.insert(make_pair(-1, "校长"));  
7 | // 第三种 通过value_type的方式插入对象  
8 | mapStu.insert(map<int, string>::value_type(1, "小李"));  
9 | // 第四种 通过数组的方式插入值  
10 | mapStu[3] = "小刘";  
11 | mapStu[5] = "小王";
```

5. 删除元素

```
1 | clear(); //删除所有元素  
2 | erase(pos); //删除pos迭代器所指的元素，返回下一个元素的迭代器。  
3 | erase(beg, end); //删除区间[beg, end)的所有元素，返回下一个元素的迭代器。  
4 | erase(keyElem); //删除容器中key为keyElem的对组。
```

6. 查找操作

```

1 find(key); //查找键key是否存在,若存在, 返回该键的元素的迭代器; /若不存在, 返回map.end();
2 count(keyElem); //返回容器中key为keyElem的对组个数。对map来说, 要么是0, 要么是1。对
3 multimap来说, 值可能大于1。
4 lower_bound(keyElem); //返回第一个key>=keyElem元素的迭代器。
5 upper_bound(keyElem); //返回第一个key>keyElem元素的迭代器。
6 equal_range(keyElem); //返回容器中key与keyElem相等的上下限的两个迭代器。

```

3.3.8.3. multimap案例

公司今天招聘了5个员工，5名员工进入公司之后，需要指派员工在那个部门工作

人员信息有：姓名 年龄 电话 工资等组成

通过Multimap进行信息的插入 保存 显示

分部门显示员工信息 显示全部员工信息

```

1 #include<iostream>
2 #include<map>
3 #include<string>
4 #include<vector>
5 using namespace std;
6
7 //multimap 案例
8 //公司今天招聘了 5 个员工, 5 名员工进入公司之后, 需要指派员工在那个部门工作
9 //人员信息有: 姓名 年龄 电话 工资等组成
10 //通过 Multimap 进行信息的插入 保存 显示
11 //分部门显示员工信息 显示全部员工信息
12
13
14 #define SALE_DEPARTMENT 1 //销售部门
15 #define DEVELOP_DEPARTMENT 2 //研发部门
16 #define FINACIAL_DEPARTMENT 3 //财务部门
17 #define ALL_DEPARTMENT 4 //所有部门
18
19 //员工类
20 class person{
21 public:
22     string name; //员工姓名
23     int age; //员工年龄
24     double salary; //员工工资
25     string tele; //员工电话
26 };
27
28 //创建5个员工
29 void CreatePerson(vector<person>& vlist){
30
31     string seed = "ABCDE";
32     for (int i = 0; i < 5; i++){

```

```

33     person p;
34     p.name = "员工";
35     p.name += seed[i];
36     p.age = rand() % 30 + 20;
37     p.salary = rand() % 20000 + 10000;
38     p.tele = "010-8888888";
39     vlist.push_back(p);
40 }
41 }
42 }
43
44 //5名员工分配到不同的部门
45 void PersonByGroup(vector<person>& vlist, multimap<int, person>& plist){
46
47
48     int operate = -1; //用户的操作
49
50     for (vector<person>::iterator it = vlist.begin(); it != vlist.end(); it++){
51
52         cout << "当前员工信息：" << endl;
53         cout << "姓名：" << it->name << " 年龄：" << it->age << " 工资：" << it-
54 >salary << " 电话：" << it->tele << endl;
55         cout << "请对该员工进行部门分配(1 销售部门, 2 研发部门, 3 财务部门)：" << endl;
56         scanf("%d", &operate);
57
58         while (true){
59
60             if (operate == SALE_DEPATMENT){ //将该员工加入到销售部门
61                 plist.insert(make_pair(SALE_DEPATMENT, *it));
62                 break;
63             }
64             else if (operate == DEVELOP_DEPATMENT){
65                 plist.insert(make_pair(DEVELOP_DEPATMENT, *it));
66                 break;
67             }
68             else if (operate == FINACIAL_DEPATMENT){
69                 plist.insert(make_pair(FINACIAL_DEPATMENT, *it));
70                 break;
71             }
72             else{
73                 cout << "您的输入有误, 请重新输入(1 销售部门, 2 研发部门, 3 财务部门)：" <<
74                 endl;
75                 scanf("%d", &operate);
76             }
77
78         }
79
80     }
81
82     cout << "员工部门分配完毕！" << endl;

```

```

80     cout << "*****" <<
81     endl;
82 }
83
84 //打印员工信息
85 void printList(multimap<int, person>& plist, int myoperate){
86
87     if (myoperate == ALL_DEPATMENT){
88         for (multimap<int, person>::iterator it = plist.begin(); it !=
89             plist.end(); it++){
90             cout << "姓名: " << it->second.name << " 年龄: " << it->second.age << "
91             工资: " << it->second.salary << " 电话: " << it->second.tele << endl;
92             }
93             return;
94     }
95
96     multimap<int, person>::iterator it = plist.find(myoperate);
97     int depatCount = plist.count(myoperate);
98     int num = 0;
99     if (it != plist.end()){
100        while (it != plist.end() && num < depatCount){
101            cout << "姓名: " << it->second.name << " 年龄: " << it->second.age << "
102            工资: " << it->second.salary << " 电话: " << it->second.tele << endl;
103            it++;
104            num++;
105        }
106    }
107
108 //根据用户操作显示不同部门的人员列表
109 void ShowPersonList(multimap<int, person>& plist, int myoperate){
110
111     switch (myoperate)
112     {
113     case SALE_DEPATMENT:
114         printList(plist, SALE_DEPATMENT);
115         break;
116     case DEVELOP_DEPATMENT:
117         printList(plist, DEVELOP_DEPATMENT);
118         break;
119     case FINACIAL_DEPATMENT:
120         printList(plist, FINACIAL_DEPATMENT);
121         break;
122     case ALL_DEPATMENT:
123         printList(plist, ALL_DEPATMENT);
124         break;
125     }

```

```

126 //用户操作菜单
127 void PersonMenue(multimap<int, person>& plist){
128
129     int flag = -1;
130     int isexit = 0;
131     while (true){
132         cout << "请输入您的操作((1 销售部门, 2 研发部门, 3 财务部门, 4 所有部门, 0退出): "
133         << endl;
134         scanf("%d", &flag);
135
136         switch (flag)
137         {
138             case SALE_DEPATMENT:
139                 ShowPersonList(plist, SALE_DEPATMENT);
140                 break;
141             case DEVELOP_DEPATMENT:
142                 ShowPersonList(plist, DEVELOP_DEPATMENT);
143                 break;
144             case FINACIAL_DEPATMENT:
145                 ShowPersonList(plist, FINACIAL_DEPATMENT);
146                 break;
147             case ALL_DEPATMENT:
148                 ShowPersonList(plist, ALL_DEPATMENT);
149                 break;
150             case 0:
151                 isexit = 1;
152                 break;
153             default:
154                 cout << "您的输入有误, 请重新输入!" << endl;
155                 break;
156         }
157
158         if (isexit == 1){
159             break;
160         }
161     }
162 }
163
164 int main(){
165
166     vector<person> vlist; //创建的5个员工 未分组
167     multimap<int, person> plist; //保存分组后员工信息
168
169     //创建5个员工
170     CreatePerson(vlist);
171     //5名员工分配到不同的部门
172     PersonByGroup(vlist, plist);
173     //根据用户输入显示不同部门员工信息列表 或者 显示全部员工的信息列表
174     PersonMenue(plist);

```

```

175
176     system("pause");
177     return EXIT_SUCCESS;
178 }
```

3.4. 算法

3.4.1. 函数对象

重载函数调用操作符的类，其对象常称为函数对象（function object），即它们是行为类似函数的对象，也叫仿函数（functor），其实就是重载“()”操作符，使得类对象可以像函数那样调用。

注意：

1. 函数对象（仿函数）是一个类，不是一个函数。
2. 函数对象（仿函数）重载了“()”操作符使得它可以像函数一样调用。

分类：假定某个类有一个重载的operator()，而且重载的operator()要求获取一个参数，我们就将这个类称为“一元仿函数”（unary functor）；相反，如果重载的operator()要求获取两个参数，就将这个类称为“二元仿函数”（binary functor）。

函数对象的作用主要是什么？STL提供的算法往往都有两个版本，其中一个版本表现出最常用的某种运算，另一版本则允许用户通过template参数的形式来指定所要采取的策略。

```

1 //函数对象是重载了函数调用符号的类
2 class MyPrint
3 {
4 public:
5     MyPrint()
6     {
7         m_Num = 0;
8     }
9     int m_Num;
10
11 public:
12     void operator() (int num)
13     {
14         cout << num << endl;
15         m_Num++;
16     }
17 };
18
19 //函数对象
20 //重载了()操作符的类实例化的对象，可以像普通函数那样调用，可以有参数，可以有返回值
21 void test01()
22 {
23     MyPrint myPrint;
24     myPrint(20);
25 }
```

```

26 }
27 // 函数对象超出了普通函数的概念，可以保存函数的调用状态
28 void test02()
29 {
30     MyPrint myPrint;
31     myPrint(20);
32     myPrint(20);
33     myPrint(20);
34     cout << myPrint.m_Num << endl;
35 }
36
37 void doBusiness(MyPrint print,int num)
38 {
39     print(num);
40 }
41
42 //函数对象作为参数
43 void test03()
44 {
45     //参数1：匿名函数对象
46     doBusiness(MyPrint(),30);
47 }
```

3.4.2. 谓语

谓词是指普通函数或重载的operator()返回值是bool类型的函数对象(仿函数)。如果operator接受一个参数，那么叫做一元谓词，如果接受两个参数，那么叫做二元谓词，谓词可作为一个判断式。

```

1 class GreaterThanFive
2 {
3 public:
4     bool operator()(int num)
5     {
6         return num > 5;
7     }
8
9 };
10 //一元谓词
11 void test01()
12 {
13     vector<int> v;
14     for (int i = 0; i < 10;i++)
15     {
16         v.push_back(i);
17     }
18
19     vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterThanFive());
20     if (it == v.end())
```

```
21     {
22         cout << "没有找到" << endl;
23     }
24     else
25     {
26         cout << "找到了: " << *it << endl;
27     }
28 }
29
30 //二元谓词
31 class MyCompare
32 {
33 public:
34     bool operator()(int num1, int num2)
35     {
36         return num1 > num2;
37     }
38 };
39
40 void test02()
41 {
42     vector<int> v;
43     v.push_back(10);
44     v.push_back(40);
45     v.push_back(20);
46     v.push_back(90);
47     v.push_back(60);
48
49     //默认从小到大
50     sort(v.begin(), v.end());
51     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
52     {
53         cout << *it << " ";
54     }
55     cout << endl;
56     cout << "-----" << endl;
57     //使用函数对象改变算法策略，排序从大到小
58     sort(v.begin(), v.end(), MyCompare());
59     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
60     {
61         cout << *it << " ";
62     }
63     cout << endl;
64 }
```

3.4.3. 内建函数对象

STL内建了一些函数对象。分为：算数类函数对象，关系运算类函数对象，逻辑运算类仿函数。这些仿函数所产生的对象，用法和一般函数完全相同，当然我们还可以产生无名的临时对象来履行函数功能。使用内建函数对象，需要引入头文件 `#include`。

6个算数类函数对象，除了`negate`是一元运算，其他都是二元运算。

```
template<*class* T> T plus//加法仿函数
template<*class* T> T minus//减法仿函数
template<*class* T> T multiplies//乘法仿函数
*template<*class* T> T divides//除法仿函数
*template<*class* T> T modulus//取模仿函数
*template<*class* T> T negate//取反仿函数
```

6个关系运算类函数对象，每一种都是二元运算。

```
*template<*class* T> bool equal_to//等于
*template<*class* T> bool not_equal_to//不等于
*template<*class* T> bool greater//大于
*template<*class* T> bool greater_equal//大于等于
*template<*class* T> bool less//小于
*template<*class* T> bool less_equal//小于等于
```

逻辑运算类运算函数，`not`为一元运算，其余为二元运算。

```
*template<*class* T> bool logical_and//逻辑与
*template<*class* T> bool logical_or//逻辑或
*template<*class* T> bool logical_not//逻辑非
```

内建函数对象举例：

```
1 //取反仿函数
2 void test01()
3 {
4     negate<int> n;
5     cout << n(50) << endl;
6 }
7
8 //加法仿函数
9 void test02()
10 {
11     plus<int> p;
12     cout << p(10, 20) << endl;
13 }
14
15 //大于仿函数
16 void test03()
17 {
18     vector<int> v;
19     srand((unsigned int)time(NULL));
20     for (int i = 0; i < 10; i++) {
```

```

21     v.push_back(rand() % 100);
22 }
23
24 for (vector<int>::iterator it = v.begin(); it != v.end(); it++){
25     cout << *it << " ";
26 }
27 cout << endl;
28 sort(v.begin(), v.end(), greater<int>());
29
30 for (vector<int>::iterator it = v.begin(); it != v.end(); it++){
31     cout << *it << " ";
32 }
33 cout << endl;
34
35 }

```

3.4.3.1. 函数对象适配器

```

1 //函数适配器bind1st bind2nd
2 //现在我有这个需求 在遍历容器的时候，我希望将容器中的值全部加上100之后显示出来，怎么做？
3 //我们直接给函数对象绑定参数 编译阶段就会报错
4 //for_each(v.begin(), v.end(), bind2nd(myprint(),100));
5 //如果我们想使用绑定适配器，需要我们自己的函数对象继承binary_function 或者 unary_function
6 //根据我们函数对象是一元函数对象 还是二元函数对象
7 class MyPrint :public binary_function<int,int,void>
8 {
9 public:
10     void operator()(int v1,int v2) const
11     {
12         cout << "v1 = :" << v1 << " v2 = :" << v2 << " v1+v2 = :" << (v1 + v2)
13         << endl;
14     }
15 };
16 //1、函数适配器
17 void test01()
18 {
19     vector<int>v;
20     for (int i = 0; i < 10; i++)
21     {
22         v.push_back(i);
23     }
24     cout << "请输入起始值: " << endl;
25     int x;
26     cin >> x;
27
28     for_each(v.begin(), v.end(), bind1st(MyPrint(), x));
29     //for_each(v.begin(), v.end(), bind2nd( MyPrint(),x ));
30 }

```

```

30 //总结: bind1st和bind2nd区别?
31 //bind1st : 将参数绑定为函数对象的第一个参数
32 //bind2nd : 将参数绑定为函数对象的第二个参数
33 //bind1st bind2nd将二元函数对象转为一元函数对象
34
35
36 class GreaterThanFive:public unary_function<int,bool>
37 {
38 public:
39     bool operator ()(int v) const
40     {
41         return v > 5;
42     }
43 };
44
45 //2、取反适配器
46 void test02()
47 {
48     vector <int> v;
49     for (int i = 0; i < 10;i++)
50     {
51         v.push_back(i);
52     }
53
54 //    vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterThanFive());
55 //    //返回第一个大于5的迭代器
56 //    vector<int>::iterator it = find_if(v.begin(), v.end(),
57 //    not1(GreaterThanFive())); //返回第一个小于5迭代器
58 //    //自定义输入
59 //    vector<int>::iterator it = find_if(v.begin(), v.end(), not1 (
60 //bind2nd(greater<int>(),5)));
61 //    if (it == v.end())
62 //    {
63 //        cout << "没找到" << endl;
64 //    }
65 //    else
66 //    {
67 //        cout << "找到" << *it << endl;
68 //    }
69
70 //排序 二元函数对象
71 sort(v.begin(), v.end(), not2(less<int>()));
72 for_each(v.begin(), v.end(), [](int val){cout << val << " "; });
73
74
75 void MyPrint03(int v,int v2)
76 {

```

```

77     cout << v + v2 << " ";
78 }
79
80 //3、函数指针适配器    ptr_fun
81 void test03()
82 {
83     vector <int> v;
84     for (int i = 0; i < 10; i++)
85     {
86         v.push_back(i);
87     }
88     // ptr_fun( )把一个普通的函数指针适配成函数对象
89     for_each(v.begin(), v.end(), bind2nd( ptr_fun( MyPrint03 ), 100));
90 }
91
92
93 //4、成员函数适配器
94 class Person
95 {
96 public:
97     Person(string name, int age)
98     {
99         m_Name = name;
100        m_Age = age;
101    }
102
103    //打印函数
104    void ShowPerson(){
105        cout << "成员函数：" << "Name：" << m_Name << " Age：" << m_Age << endl;
106    }
107    void Plus100()
108    {
109        m_Age += 100;
110    }
111 public:
112     string m_Name;
113     int m_Age;
114 };
115
116 void MyPrint04(Person &p)
117 {
118     cout << "姓名：" << p.m_Name << " 年龄：" << p.m_Age << endl;
119 }
120
121
122 void test04()
123 {
124     vector <Person>v;
125     Person p1("aaa", 10);
126     Person p2("bbb", 20);

```

```

127     Person p3("ccc", 30);
128     Person p4("ddd", 40);
129     v.push_back(p1);
130     v.push_back(p2);
131     v.push_back(p3);
132     v.push_back(p4);
133
134     //for_each(v.begin(), v.end(), MyPrint04);
135     //利用 mem_fun_ref 将Person内部成员函数适配
136     for_each(v.begin(), v.end(), mem_fun_ref(&Person::ShowPerson));
137 //    for_each(v.begin(), v.end(), mem_fun_ref(&Person::Plus100));
138 //    for_each(v.begin(), v.end(), mem_fun_ref(&Person::ShowPerson));
139 }
140
141 void test05(){
142
143     vector<Person*> v1;
144     //创建数据
145     Person p1("aaa", 10);
146     Person p2("bbb", 20);
147     Person p3("ccc", 30);
148     Person p4("ddd", 40);
149
150     v1.push_back(&p1);
151     v1.push_back(&p2);
152     v1.push_back(&p3);
153     v1.push_back(&p4);
154
155     for_each(v1.begin(), v1.end(), mem_fun(&Person::ShowPerson));
156 }
157
158 //如果容器存放的是对象指针， 那么用mem_fun
159 //如果容器中存放的是对象实体， 那么用mem_fun_ref

```

3.4.4. 算法概述

算法主要是由头文件组成。

是所有STL头文件中最大的一个,其中常用的功能涉及到比较,交换,查找,遍历,复制,修改,反转,排序,合并等...

体积很小,只包括在几个序列容器上进行的简单运算的模板函数.

定义了一些模板类,用以声明函数对象。

3.4.5. 常用遍历算法

3.4.5.1. for_each遍历算法

```

1  /*
2   * 遍历算法 遍历容器元素
3   * @param beg 开始迭代器
4   * @param end 结束迭代器
5   * @param _callback 函数回调或者函数对象
6   * @return 函数对象
7   */
8  for_each(iterator beg, iterator end, _callback);

```

```

1  /*template<class _InIt, class _Fn1> inline
2  void for_each(_InIt _First, _InIt _Last, _Fn1 _Func)
3  {
4      for (; _First != _Last; ++_First)
5          _Func(*_First);
6  }
7  */
8
9 //普通函数
10 void print01(int val){
11     cout << val << " ";
12 }
13 //函数对象
14 struct print01{
15     void operator()(int val){
16         cout << val << " ";
17     }
18 };
19
20 //for_each算法基本用法
21 void test01(){
22
23
24 vector<int> v;
25 for (int i = 0; i < 10;i++){
26     v.push_back(i);
27 }
28
29 //遍历算法
30 for_each(v.begin(), v.end(), print01);
31 cout << endl;
32
33 for_each(v.begin(), v.end(), print01());
34 cout << endl;
35
36 }

```

```
37
38 struct print02{
39     print02(){
40         mCount = 0;
41     }
42     void operator()(int val){
43         cout << val << " ";
44         mCount++;
45     }
46     int mCount;
47 };
48
49 //for_each返回值
50 void test02(){
51
52     vector<int> v;
53     for (int i = 0; i < 10; i++){
54         v.push_back(i);
55     }
56
57     print02 p = for_each(v.begin(), v.end(), print02());
58     cout << endl;
59     cout << p.mCount << endl;
60
61 }
62
63 struct print03 : public binary_function<int, int, void>{
64     void operator()(int val,int bindParam) const{
65         cout << val + bindParam << " ";
66     }
67 };
68
69 //for_each绑定参数输出
70 void test03(){
71
72     vector<int> v;
73     for (int i = 0; i < 10; i++){
74         v.push_back(i);
75     }
76
77     for_each(v.begin(), v.end(), bind2nd(print03(),100));
78
79 }
80 }
```

3.4.5.2. transform算法

```

1  /*
2   * transform 算法 将指定容器区间元素搬运到另一容器中
3   * 注意 : transform 不会给目标容器分配内存, 所以需要我们提前分配好内存
4   * @param beg1 源容器开始迭代器
5   * @param end1 源容器结束迭代器
6   * @param beg2 目标容器开始迭代器
7   * @param _cakback 回调函数或者函数对象
8   * @return 返回目标容器迭代器
9  */
10 transform(iterator beg1, iterator end1, iterator beg2, _callbakc);

```

```

1 //transform 将一个容器中的值搬运到另一个容器中
2 /*
3  template<class _InIt, class _OutIt, class _Fn1> inline
4  _OutIt _Transform(_InIt _First, _InIt _Last,_OutIt _Dest, _Fn1 _Func)
5  {
6
7      for (; _First != _Last; ++_First, ++_Dest)
8          *_Dest = _Func(*_First);
9      return (_Dest);
10 }
11
12 template<class _InIt1,class _InIt2,class _OutIt,class _Fn2> inline
13 _OutIt _Transform(_InIt1 _First1, _InIt1 _Last1,_InIt2 _First2, _OutIt _Dest,
14 _Fn2 _Func)
15 {
16     for (; _First1 != _Last1; ++_First1, ++_First2, ++_Dest)
17         *_Dest = _Func(*_First1, *_First2);
18     return (_Dest);
19 }
20
21 struct transformTest01{
22     int operator()(int val){
23         return val + 100;
24     }
25 };
26 struct print01{
27     void operator()(int val){
28         cout << val << " ";
29     }
30 };
31 void test01(){
32
33     vector<int> vSource;
34     for (int i = 0; i < 10;i ++){
35         vSource.push_back(i + 1);

```

```

36 }
37
38 //目标容器
39 vector<int> vTarget;
40 //给vTarget开辟空间
41 vTarget.resize(vSource.size());
42 //将vSource中的元素搬运到vTarget
43 vector<int>::iterator it = transform(vSource.begin(), vSource.end(),
44 vTarget.begin(), transformTest01());
45 //打印
46 for_each(vTarget.begin(), vTarget.end(), print01()); cout << endl;
47 }
48
49 //将容器1和容器2中的元素相加放入到第三个容器中
50 struct transformTest02{
51     int operator()(int v1,int v2){
52         return v1 + v2;
53     }
54 };
55 void test02(){
56
57     vector<int> vSource1;
58     vector<int> vSource2;
59     for (int i = 0; i < 10; i++){
60         vSource1.push_back(i + 1);
61     }
62
63     //目标容器
64     vector<int> vTarget;
65     //给vTarget开辟空间
66     vTarget.resize(vSource1.size());
67     transform(vSource1.begin(), vSource1.end(), vSource2.begin(),vTarget.begin(),
68     transformTest02());
69     //打印
70     for_each(vTarget.begin(), vTarget.end(), print01()); cout << endl;
}

```

3.4.6. 常用查找算法

```

1 /*
2  find算法 查找元素
3  @param beg 容器开始迭代器
4  @param end 容器结束迭代器
5  @param value 查找的元素
6  @return 返回查找元素的位置
7 */
8 find(iterator beg, iterator end, value)

```

```
9  /*
10   *      find_if算法 条件查找
11   *      @param beg 容器开始迭代器
12   *      @param end 容器结束迭代器
13   *      @param callback 回调函数或者谓词(返回bool类型的函数对象)
14   *      @return bool 查找返回true 否则false
15   */
16 find_if(iterator beg, iterator end, _callback);
17
18 /*
19   *      adjacent_find算法 查找相邻重复元素
20   *      @param beg 容器开始迭代器
21   *      @param end 容器结束迭代器
22   *      @param _callback 回调函数或者谓词(返回bool类型的函数对象)
23   *      @return 返回相邻元素的第一个位置的迭代器
24   */
25 adjacent_find(iterator beg, iterator end, _callback);
26 /*
27   *      binary_search算法 二分查找法
28   *      注意: 在无序序列中不可用
29   *      @param beg 容器开始迭代器
30   *      @param end 容器结束迭代器
31   *      @param value 查找的元素
32   *      @return bool 查找返回true 否则false
33   */
34 bool binary_search(iterator beg, iterator end, value);
35 /*
36   *      count算法 统计元素出现次数
37   *      @param beg 容器开始迭代器
38   *      @param end 容器结束迭代器
39   *      @param value回调函数或者谓词(返回bool类型的函数对象)
40   *      @return int返回元素个数
41   */
42 count(iterator beg, iterator end, value);
43 /*
44   *      count_if算法 统计元素出现次数
45   *      @param beg 容器开始迭代器
46   *      @param end 容器结束迭代器
47   *      @param callback 回调函数或者谓词(返回bool类型的函数对象)
48   *      @return int返回元素个数
49   */
50 count_if(iterator beg, iterator end, _callback);
```

3.4.7. 常用排序算法

```

1  /*
2   * merge算法 容器元素合并，并存储到另一容器中
3   * 注意：两个容器必须是有序的
4   * @param beg1 容器1开始迭代器
5   * @param end1 容器1结束迭代器
6   * @param beg2 容器2开始迭代器
7   * @param end2 容器2结束迭代器
8   * @param dest 目标容器开始迭代器
9  */
10 merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
11 /*
12  sort算法 容器元素排序
13  @param beg 容器1开始迭代器
14  @param end 容器1结束迭代器
15  @param _callback 回调函数或者谓词(返回bool类型的函数对象)
16 */
17 sort(iterator beg, iterator end, _callback)
18 /*
19  random_shuffle算法 对指定范围内的元素随机调整次序
20  @param beg 容器开始迭代器
21  @param end 容器结束迭代器
22 */
23 random_shuffle(iterator beg, iterator end)
24 /*
25  reverse算法 反转指定范围的元素
26  @param beg 容器开始迭代器
27  @param end 容器结束迭代器
28 */
29 reverse(iterator beg, iterator end)

```

3.4.8. 常用拷贝和替换算法

```

1  /*
2   * copy算法 将容器内指定范围的元素拷贝到另一容器中
3   * @param beg 容器开始迭代器
4   * @param end 容器结束迭代器
5   * @param dest 目标起始迭代器
6  */
7  copy(iterator beg, iterator end, iterator dest)
8  /*
9   * replace算法 将容器内指定范围的旧元素修改为新元素
10  @param beg 容器开始迭代器
11  @param end 容器结束迭代器
12  @param oldvalue 旧元素
13  @param newvalue 新元素

```

```

14 */
15 replace(iterator beg, iterator end, oldvalue, newvalue)
16 /*
17     replace_if算法 将容器内指定范围满足条件的元素替换为新元素
18     @param beg 容器开始迭代器
19     @param end 容器结束迭代器
20     @param callback函数回调或者谓词(返回Bool类型的函数对象)
21     @param oldvalue 新元素
22 */
23 replace_if(iterator beg, iterator end, _callback, newvalue)
24 /*
25     swap算法 互换两个容器的元素
26     @param c1容器1
27     @param c2容器2
28 */
29 swap(container c1, container c2)

```

3.4.9. 常用算数生成算法

```

1 /*
2     accumulate算法 计算容器元素累计总和
3     @param beg 容器开始迭代器
4     @param end 容器结束迭代器
5     @param value累加值
6 */
7 accumulate(iterator beg, iterator end, value)
8 /*
9     fill算法 向容器中添加元素
10    @param beg 容器开始迭代器
11    @param end 容器结束迭代器
12    @param value t填充元素
13 */
14 fill(iterator beg, iterator end, value)

```

3.4.10. 常用集合算法

```

1 /*
2     set_intersection算法 求两个set集合的交集
3     注意:两个集合必须是有序序列
4     @param beg1 容器1开始迭代器
5     @param end1 容器1结束迭代器
6     @param beg2 容器2开始迭代器
7     @param end2 容器2结束迭代器
8     @param dest 目标容器开始迭代器
9     @return 目标容器的最后一个元素的迭代器地址

```

```
10  */
11 set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2,
12 iterator dest)
13 /*
14     set_union算法 求两个set集合的并集
15     注意:两个集合必须是有序序列
16     @param beg1 容器1开始迭代器
17     @param end1 容器1结束迭代器
18     @param beg2 容器2开始迭代器
19     @param end2 容器2结束迭代器
20     @param dest 目标容器开始迭代器
21     @return 目标容器的最后一个元素的迭代器地址
22 */
23 set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator
24 dest)
25 /*
26     set_difference算法 求两个set集合的差集
27     注意:两个集合必须是有序序列
28     @param beg1 容器1开始迭代器
29     @param end1 容器1结束迭代器
30     @param beg2 容器2开始迭代器
31     @param end2 容器2结束迭代器
32     @param dest 目标容器开始迭代器
33     @return 目标容器的最后一个元素的迭代器地址
34 */
35 set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2,
36 iterator dest)
```