

# C++基础入门

---

## 1 C++初识

---

### 1.1 第一个C++程序

编写一个C++程序总共分为4个步骤

- 创建项目
- 创建文件
- 编写代码
- 运行程序

#### 1.1.1 创建项目

Visual Studio是我们用来编写C++程序的主要工具，我们先将它打开

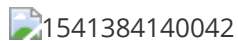


#### 1.1.2 创建文件

右键源文件，选择添加->新建项



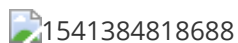
给C++文件起个名称，然后点击添加即可。



#### 1.1.3 编写代码

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5
6      cout << "Hello world" << endl;
7
8      system("pause");
9
10     return 0;
11 }
```

#### 1.1.4 运行程序



## 1.2 注释

**作用：**在代码中加一些说明和解释，方便自己或其他程序员程序员阅读代码

**两种格式**

1. **单行注释：** `// 描述信息`

- 通常放在一行代码的上方，或者一条语句的末尾，**对该行代码说明**

2. **多行注释：** `/* 描述信息 */`

- 通常放在一段代码的上方，**对该段代码做整体说明**

提示：编译器在编译代码时，会忽略注释的内容

**ctrl + K + C可以直接注释多行代码**

## 1.3 变量

**作用：**给一段指定的内存空间起名，方便操作这段内存

**语法：** `数据类型 变量名 = 初始值;`

**示例：**

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5
6      //变量的定义
7      //语法：数据类型 变量名 = 初始值
8
9      int a = 10;
10
11     cout << "a = " << a << endl;
12
13     system("pause");
14
15     return 0;
16 }
```

注意：C++在创建变量时，必须给变量一个初始值，否则会报错

## 1.4 常量

**作用：**用于记录程序中不可更改的数据

C++定义常量两种方式

1. **#define** 宏常量： `#define 常量名 常量值`
  - 通常在文件上方定义，表示一个常量
2. **const**修饰的变量 `const 数据类型 常量名 = 常量值`
  - 通常在变量定义前加关键字**const**，修饰该变量为常量，不可修改

**示例：**

```
1 //1、宏常量
2 #define day 7
3
4 int main() {
5
6     cout << "一周里总共有 " << day << " 天" << endl;
7     //day = 8; //报错，宏常量不可以修改
8
9     //2、const修饰变量
10    const int month = 12;
11    cout << "一年里总共有 " << month << " 个月份" << endl;
12    //month = 24; //报错，常量是不可以修改的
13
14
15    system("pause");
16
17    return 0;
18 }
```

## 1.5 关键字

**作用：**关键字是C++中预先保留的单词（标识符）

- 在定义变量或者常量时候，不要用关键字

C++关键字如下：

asm	do	if	return	typedef
auto	double	inline	short	typeid
bool	dynamic_cast	int	signed	typename
break	else	long	sizeof	union
case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using
char	export	new	struct	virtual
class	extern	operator	switch	void
const	false	private	template	volatile
const_cast	float	protected	this	wchar_t
continue	for	public	throw	while
default	friend	register	true	
delete	goto	reinterpret_cast	try	

提示：在给变量或者常量起名称时候，不要用C++得关键字，否则会产生歧义。

## 1.6 标识符命名规则

**作用：**C++规定给标识符（变量、常量）命名时，有一套自己的规则

- 标识符不能是关键字
- 标识符只能由字母、数字、下划线组成
- 第一个字符必须为字母或下划线
- 标识符中字母区分大小写

建议：给标识符命名时，争取做到见名知意的效果，方便自己和他人的阅读

## 2 数据类型

C++规定在创建一个变量或者常量时，必须要指定出相应的数据类型，否则无法给变量分配内存

### 2.1 整型

作用：整型变量表示的是整数类型的数据

C++中能够表示整型的类型有以下几种方式，区别在于所占内存空间不同：

数据类型	占用空间	取值范围
short(短整型)	2字节	$(-2^{15} \sim 2^{15}-1)$
int(整型)	4字节	$(-2^{31} \sim 2^{31}-1)$
long(长整形)	Windows为4字节，Linux为4字节(32位)，8字节(64位)	$(-2^{31} \sim 2^{31}-1)$
long long(长长整形)	8字节	$(-2^{63} \sim 2^{63}-1)$

### 2.2 sizeof关键字

作用：利用sizeof关键字可以统计数据类型所占内存大小

语法：sizeof(数据类型 / 变量)

示例：

```
1 int main() {
2
3     cout << "short 类型所占内存空间为: " << sizeof(short) << endl;
4
5     cout << "int 类型所占内存空间为: " << sizeof(int) << endl;
6 }
```

```

7      cout << "long 类型所占内存空间为: " << sizeof(long) << endl;
8
9      cout << "long long 类型所占内存空间为: " << sizeof(long long) << endl;
10
11     system("pause");
12
13     return 0;
14 }

```

整型结论: `short < int <= long <= long long`

## 2.3 实型（浮点型）

作用: 用于表示小数

浮点型变量分为两种:

1. 单精度float
2. 双精度double

两者的区别在于表示的有效数字范围不同。

数据类型	占用空间	有效数字范围
float	4字节	7位有效数字
double	8字节	15~16位有效数字

示例:

```

1  int main() {
2
3      float f1 = 3.14f;
4      double d1 = 3.14;
5
6      cout << f1 << endl;
7      cout << d1 << endl;
8
9      cout << "float  sizeof = " << sizeof(f1) << endl;
10     cout << "double sizeof = " << sizeof(d1) << endl;
11
12     //科学计数法
13     float f2 = 3e2; // 3 * 10 ^ 2
14     cout << "f2 = " << f2 << endl;
15

```

```

16     float f3 = 3e-2; // 3 * 0.1 ^ 2
17     cout << "f3 = " << f3 << endl;
18
19     system("pause");
20
21     return 0;
22 }

```

## 2.4 字符型

**作用：**字符型变量用于显示单个字符

**语法：** `char ch = 'a';`

注意1：在显示字符型变量时，用单引号将字符括起来，不要用双引号

注意2：单引号内只能有一个字符，不可以是字符串

- C和C++中字符型变量只占用1个字节。
- 字符型变量并不是把字符本身放到内存中存储，而是将对应的ASCII编码放入到存储单元

示例：

```

1  int main() {
2
3      char ch = 'a';
4      cout << ch << endl;
5      cout << sizeof(char) << endl;
6
7      //ch = "abcde"; //错误，不可以用双引号
8      //ch = 'abcde'; //错误，单引号内只能引用一个字符
9
10     cout << (int)ch << endl; //查看字符a对应的ASCII码
11     ch = 97; //可以直接用ASCII给字符型变量赋值
12     cout << ch << endl;
13
14     system("pause");
15
16     return 0;
17 }

```

ASCII码表格：

ASCII值	控制字符	ASCII值	字符	ASCII值	字符	ASCII值	字符
0	NUT	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	/	124	
29	GS	61	=	93	]	125	}



ASCII值	控制字符	ASCII值	字符	ASCII值	字符	ASCII值	字符
30	RS	62	>	94	^	126	`
31	US	63	?	95	_	127	DEL

ASCII 码大致由以下**两部分**组成：

- ASCII 非打印控制字符：ASCII 表上的数字 **0-31** 分配给了控制字符，用于控制像打印机等一些外围设备。
- ASCII 打印字符：数字 **32-126** 分配给了能在键盘上找到的字符，当查看或打印文档时就会出现。

## 2.5 转义字符

**作用：**用于表示一些不能显示出来的ASCII字符

现阶段我们常用的转义字符有：`\n` `\\` `\t`

转义字符	含义	ASCII码值（十进制）
<code>\a</code>	警报	007
<code>\b</code>	退格(BS)，将当前位置移到前一位	008
<code>\f</code>	换页(FF)，将当前位置移到下页开头	012
<code>\n</code>	<b>换行(LF)，将当前位置移到下一行开头</b>	<b>010</b>
<code>\r</code>	回车(CR)，将当前位置移到本行开头	013
<code>\t</code>	<b>水平制表(HT)（跳到下一个TAB位置）</b>	<b>009</b>
<code>\v</code>	垂直制表(VT)	011
<code>\\</code>	<b>代表一个反斜线字符"</b>	<b>092</b>
<code>'</code>	代表一个单引号（撇号）字符	039
<code>"</code>	代表一个双引号字符	034
<code>\?</code>	代表一个问号	063
<code>\0</code>	数字0	000
<code>\ddd</code>	8进制转义字符，d范围0~7	3位8进制
<code>\xhh</code>	16进制转义字符，h范围0 <sub>9</sub> , a <sub>f</sub> , A~F	3位16进制

示例：

```

1  int main() {
2
3
4      cout << "\\\" << endl;
5      cout << "\\tHello" << endl;
6      cout << "\\n" << endl;
7
8      system("pause");
9
10     return 0;
11 }

```

## 2.6 字符串型

作用：用于表示一串字符

两种风格

1. **C风格字符串**： `char 变量名[] = "字符串值"`

示例：

```

1  int main() {
2
3      char str1[] = "hello world";
4      cout << str1 << endl;
5
6      system("pause");
7
8      return 0;
9  }

```

注意：C风格的字符串要用双引号括起来

1. **C++风格字符串**： `string 变量名 = "字符串值"`

示例：

```

1  int main() {
2
3      string str = "hello world";
4      cout << str << endl;
5
6      system("pause");
7
8      return 0;
9  }

```

注意：C++风格字符串，需要加入头文件 `#include<string>`

## 2.7 布尔类型 bool

**作用：**布尔数据类型代表真或假的值

bool类型只有两个值：

- true --- 真（本质是1）
- false --- 假（本质是0）

**bool类型占1个字节大小**

示例：

```
1  int main() {
2
3      bool flag = true;
4      cout << flag << endl; // 1
5
6      flag = false;
7      cout << flag << endl; // 0
8
9      cout << "size of bool = " << sizeof(bool) << endl; //1
10
11     system("pause");
12
13     return 0;
14 }
```

- 布尔数据类型只要不为0都为真

## 2.8 数据的输入

**作用：**用于从键盘获取数据

**关键字：**cin

**语法：** `cin >> 变量`

示例：

```
1  int main(){
2
3      //整型输入
4      int a = 0;
5      cout << "请输入整型变量: " << endl;
6      cin >> a;
7      cout << a << endl;
8
9      //浮点型输入
10     double d = 0;
11     cout << "请输入浮点型变量: " << endl;
12     cin >> d;
13     cout << d << endl;
14
15     //字符型输入
```

```
16     char ch = 0;
17     cout << "请输入字符型变量: " << endl;
18     cin >> ch;
19     cout << ch << endl;
20
21     //字符串型输入
22     string str;
23     cout << "请输入字符串型变量: " << endl;
24     cin >> str;
25     cout << str << endl;
26
27     //布尔类型输入
28     bool flag = true;
29     cout << "请输入布尔型变量: " << endl;
30     cin >> flag;
31     cout << flag << endl;
32     system("pause");
33     return EXIT_SUCCESS;
34 }
```

## 3 运算符

**作用：**用于执行代码的运算

本章我们主要讲解以下几类运算符：

运算符类型	作用
算术运算符	用于处理四则运算
赋值运算符	用于将表达式的值赋给变量
比较运算符	用于表达式的比较，并返回一个真值或假值
逻辑运算符	用于根据表达式的值返回真值或假值

### 3.1 算术运算符

**作用：**用于处理四则运算

算术运算符包括以下符号：

运算符	术语	示例	结果
+	正号	+3	3
-	负号	-3	-3
+	加	10 + 5	15
-	减	10 - 5	5
*	乘	10 * 5	50
/	除	10 / 5	2
%	取模(取余)	10 % 3	1
++	前置递增	a=2; b=++a;	a=3; b=3;
++	后置递增	a=2; b=a++;	a=3; b=2;
--	前置递减	a=2; b=--a;	a=1; b=1;
--	后置递减	a=2; b=a--;	a=1; b=2;

### 示例1:

```

1 //加减乘除
2 int main() {
3
4     int a1 = 10;
5     int b1 = 3;
6
7     cout << a1 + b1 << endl;
8     cout << a1 - b1 << endl;
9     cout << a1 * b1 << endl;
10    cout << a1 / b1 << endl; //两个整数相除结果依然是整数，会将小数部分去除掉
11
12    int a2 = 10;
13    int b2 = 20;
14    cout << a2 / b2 << endl;
15
16    int a3 = 10;
17    int b3 = 0;
18    //cout << a3 / b3 << endl; //报错，除数不可以为0
19
20
21    //两个小数可以相除
22    double d1 = 0.5;
23    double d2 = 0.25;
24    cout << d1 / d2 << endl;
25
26    system("pause");
27
28    return 0;
29 }
```

总结：在除法运算中，除数不能为0

### 示例2:

```
1 //取模
2 int main() {
3
4     int a1 = 10;
5     int b1 = 3;
6
7     cout << 10 % 3 << endl;
8
9     int a2 = 10;
10    int b2 = 20;
11
12    cout << a2 % b2 << endl;
13
14    int a3 = 10;
15    int b3 = 0;
16
17    //cout << a3 % b3 << endl; //取模运算时，除数也不能为0
18
19    //两个小数不可以取模
20    double d1 = 3.14;
21    double d2 = 1.1;
22
23    //cout << d1 % d2 << endl;
24
25    system("pause");
26
27    return 0;
28 }
29
```

总结：只有整型变量可以进行取模运算，取模运算是基于除法运算的，所以除数为0

### 示例3:

```
1 //递增
2 int main() {
3
4     //后置递增
5     int a = 10;
6     a++; //等价于a = a + 1
7     cout << a << endl; // 11
8
9     //前置递增
10    int b = 10;
11    ++b;
12    cout << b << endl; // 11
13
14    //区别
15    //前置递增先对变量进行++，再计算表达式
16    int a2 = 10;
```

```

17     int b2 = ++a2 * 10;
18     cout << b2 << endl;
19
20     //后置递增先计算表达式，后对变量进行++
21     int a3 = 10;
22     int b3 = a3++ * 10;
23     cout << b3 << endl;
24
25     system("pause");
26
27     return 0;
28 }
29

```

总结：前置递增先对变量进行++，再计算表达式，后置递增相反

## 3.2 赋值运算符

**作用：**用于将表达式的值赋给变量

赋值运算符包括以下几个符号：

运算符	术语	示例	结果
=	赋值	a=2; b=3;	a=2; b=3;
+=	加等于	a=0; a+=2;	a=2;
-=	减等于	a=5; a-=3;	a=2;
*=	乘等于	a=2; a*=2;	a=4;
/=	除等于	a=4; a/=2;	a=2;
%=	模等于	a=3; a%=2;	a=1;

**示例：**

```

1  int main() {
2
3      //赋值运算符
4
5      // =
6      int a = 10;
7      a = 100;
8      cout << "a = " << a << endl;
9

```

```

10 // +=
11 a = 10;
12 a += 2; // a = a + 2;
13 cout << "a = " << a << endl;
14
15 // -=
16 a = 10;
17 a -= 2; // a = a - 2
18 cout << "a = " << a << endl;
19
20 // *=
21 a = 10;
22 a *= 2; // a = a * 2
23 cout << "a = " << a << endl;
24
25 // /=
26 a = 10;
27 a /= 2; // a = a / 2;
28 cout << "a = " << a << endl;
29
30 // %=
31 a = 10;
32 a %= 2; // a = a % 2;
33 cout << "a = " << a << endl;
34
35 system("pause");
36
37 return 0;
38 }

```

### 3.3 比较运算符

**作用：**用于表达式的比较，并返回一个真值或假值

比较运算符有以下符号：

运算符	术语	示例	结果
==	相等	4 == 3	0
!=	不等	4 != 3	1
<	小于	4 < 3	0
>	大于	4 > 3	1
<=	小于等于	4 <= 3	0
>=	大于等于	4 >= 1	1

示例：



```

1  int main() {
2
3      int a = 10;
4      int b = 20;
5
6      cout << (a == b) << endl; // 0 //括号的使用保证了运算的优先级
7
8      cout << (a != b) << endl; // 1
9
10     cout << (a > b) << endl; // 0
11
12     cout << (a < b) << endl; // 1
13
14     cout << (a >= b) << endl; // 0
15
16     cout << (a <= b) << endl; // 1
17
18     system("pause");
19
20     return 0;
21 }

```

注意：C和C++ 语言的比较运算中，“真”用数字“1”来表示，“假”用数字“0”来表示。

### 3.4 逻辑运算符

**作用：**用于根据表达式的值返回真值或假值

逻辑运算符有以下符号：

运算符	术语	示例	结果
!	非	!a	如果a为假，则!a为真；如果a为真，则!a为假。
&&	与	a && b	如果a和b都为真，则结果为真，否则为假。
	或	a    b	如果a和b有一个为真，则结果为真，二者都为假时，结果为假。

**示例1：**逻辑非

```

1 //逻辑运算符 --- 非
2 int main() {
3
4     int a = 10;
5
6     cout << !a << endl; // 0
7
8     cout << !!a << endl; // 1
9
10    system("pause");
11
12    return 0;
13 }

```

总结：真变假，假变真

## 示例2: 逻辑与

```

1 //逻辑运算符 --- 与
2 int main() {
3
4     int a = 10;
5     int b = 10;
6
7     cout << (a && b) << endl; // 1
8
9     a = 10;
10    b = 0;
11
12    cout << (a && b) << endl; // 0
13
14    a = 0;
15    b = 0;
16
17    cout << (a && b) << endl; // 0
18
19    system("pause");
20
21    return 0;
22 }
23

```

总结：逻辑与运算符总结：同真为真，其余为假

## 示例3: 逻辑或

```

1 //逻辑运算符 --- 或
2 int main() {

```

```

3
4     int a = 10;
5     int b = 10;
6
7     cout << (a || b) << endl; // 1
8
9     a = 10;
10    b = 0;
11
12    cout << (a || b) << endl; // 1
13
14    a = 0;
15    b = 0;
16
17    cout << (a || b) << endl; // 0
18
19    system("pause");
20
21    return 0;
22 }

```

逻辑或运算符总结：同假为假，其余为真

## 4 程序流程结构

C/C++支持最基本的三种程序运行结构：顺序结构、选择结构、循环结构

- 顺序结构：程序按顺序执行，不发生跳转
- 选择结构：依据条件是否满足，有选择的执行相应功能
- 循环结构：依据条件是否满足，循环多次执行某段代码

### 4.1 选择结构

#### 4.1.1 if语句

**作用：**执行满足条件的语句

if语句的三种形式

- 单行格式if语句
- 多行格式if语句
- 多条件的if语句

1. 单行格式if语句: `if(条件){ 条件满足执行的语句 }`



示例:

```
1  int main() {
2
3      //选择结构-单行if语句
4      //输入一个分数，如果分数大于600分，视为考上一本大学，并在屏幕上打印
5
6      int score = 0;
7      cout << "请输入一个分数: " << endl;
8      cin >> score;
9
10     cout << "您输入的分数为: " << score << endl;
11
12     //if语句
13     //注意事项，在if判断语句后面，不要加分号
14     if (score > 600)
15     {
16         cout << "我考上了一本大学!!! " << endl;
17     }
18
19     system("pause");
20
21     return 0;
22 }
```

注意: if条件表达式后不要加分号

2. 多行格式if语句: `if(条件){ 条件满足执行的语句 }else{ 条件不满足执行的语句 };`



示例:

```
1  int main() {
2
3      int score = 0;
4
5      cout << "请输入考试分数: " << endl;
6
7      cin >> score;
8
9      if (score > 600)
10     {
11         cout << "我考上了一本大学" << endl;
12     }
13 }
```

```

12     }
13     else
14     {
15         cout << "我未考上一本大学" << endl;
16     }
17
18     system("pause");
19
20     return 0;
21 }

```

3. 多条件的if语句: `if(条件1){ 条件1满足执行的语句 }else if(条件2){条件2满足执行的语句}... else{ 都不满足执行的语句}`



示例:

```

1     int main() {
2
3     int score = 0;
4
5     cout << "请输入考试分数: " << endl;
6
7     cin >> score;
8
9     if (score > 600)
10    {
11        cout << "我考上了一本大学" << endl;
12    }
13    else if (score > 500)
14    {
15        cout << "我考上了二本大学" << endl;
16    }
17    else if (score > 400)
18    {
19        cout << "我考上了三本大学" << endl;
20    }
21    else
22    {
23        cout << "我未考上本科" << endl;
24    }
25
26    system("pause");

```

```
27  
28     return 0;  
29 }
```

**嵌套if语句：**在if语句中，可以嵌套使用if语句，达到更精确的条件判断

案例需求：

- 提示用户输入一个高考考试分数，根据分数做如下判断
- 分数如果大于600分视为考上一本，大于500分考上二本，大于400考上三本，其余视为未考上本科；
- 在一本分数中，如果大于700分，考入北大，大于650分，考入清华，大于600考入人大。

示例：

```
1  int main() {  
2  
3      int score = 0;  
4  
5      cout << "请输入考试分数: " << endl;  
6  
7      cin >> score;  
8  
9      if (score > 600)  
10     {  
11         cout << "我考上了一本大学" << endl;  
12         if (score > 700)  
13         {  
14             cout << "我考上了北大" << endl;  
15         }  
16         else if (score > 650)  
17         {  
18             cout << "我考上了清华" << endl;  
19         }  
20         else  
21         {  
22             cout << "我考上了人大" << endl;  
23         }  
24     }  
25  
26     else if (score > 500)  
27     {  
28         cout << "我考上了二本大学" << endl;  
29     }  
30     else if (score > 400)  
31     {  
32         cout << "我考上了三本大学" << endl;  
33     }
```

```

33     }
34     else
35     {
36         cout << "我未考上本科" << endl;
37     }
38
39     system("pause");
40
41     return 0;
42 }

```

### 练习案例：三只小猪称体重

有三只小猪ABC，请分别输入三只小猪的体重，并且判断哪只小猪最重？  三只小猪

## 4.1.2 三目运算符

**作用：**通过三目运算符实现简单的判断

**语法：**表达式1 ? 表达式2 : 表达式3

**解释：**

如果表达式1的值为真，执行表达式2，并返回表达式2的结果；

如果表达式1的值为假，执行表达式3，并返回表达式3的结果。

**示例：**

```

1  int main() {
2
3      int a = 10;
4      int b = 20;
5      int c = 0;
6
7      c = a > b ? a : b;
8      cout << "c = " << c << endl;
9
10     //C++中三目运算符返回的是变量,可以继续赋值
11
12     (a > b ? a : b) = 100;
13
14     cout << "a = " << a << endl;
15     cout << "b = " << b << endl;
16     cout << "c = " << c << endl;
17
18     system("pause");

```

```
19
20     return 0;
21 }
```

总结：和if语句比较，三目运算符优点是短小整洁，缺点是如果用嵌套，结构不清晰

### 4.1.3 switch语句

**作用：**执行多条件分支语句

**语法：**

```
1  switch(表达式)
2
3  {
4
5      case 结果1: 执行语句;break;
6
7      case 结果2: 执行语句;break;
8
9      ...
10
11     default:执行语句;break;
12
13 }
14
```

**示例：**

```
1  int main() {
2
3      //请给电影评分
4      //10 ~ 9    经典
5      // 8 ~ 7    非常好
6      // 6 ~ 5    一般
7      // 5分以下 烂片
8
9      int score = 0;
10     cout << "请给电影打分" << endl;
11     cin >> score;
12
13     switch (score)
14     {
15     case 10:
16     case 9:
```



```

17         cout << "经典" << endl;
18         break;
19     case 8:
20         cout << "非常好" << endl;
21         break;
22     case 7:
23     case 6:
24         cout << "一般" << endl;
25         break;
26     default:
27         cout << "烂片" << endl;
28         break;
29     }
30
31     system("pause");
32
33     return 0;
34 }

```

注意1：switch语句中表达式类型只能是整型或者字符型

注意2：case里如果没有break，那么程序会一直向下执行

总结：与if语句比，对于多条件判断时，switch的结构清晰，执行效率高，缺点是switch不可以判断区间

## 4.2 循环结构

### 4.2.1 while循环语句

**作用：**满足循环条件，执行循环语句

**语法：** `while(循环条件){ 循环语句 }`

**解释：**只要循环条件的结果为真，就执行循环语句



**示例：**

```

1  int main() {
2
3      int num = 0;
4      while (num < 10)
5      {
6          cout << "num = " << num << endl;
7          num++;
8      }
9
10     system("pause");
11
12     return 0;
13 }

```

注意：在执行循环语句时候，程序必须提供跳出循环的出口，否则出现死循环

#### while循环练习案例：猜数字

**案例描述：**系统随机生成一个1到100之间的数字，玩家进行猜测，如果猜错，提示玩家数字过大或过小，如果猜对恭喜玩家胜利，并且退出游戏。



#### 4.2.2 do...while循环语句

**作用：**满足循环条件，执行循环语句

**语法：** `do{ 循环语句 } while(循环条件);`

**注意：**与while的区别在于do...while会先执行一次循环语句，再判断循环条件



**示例：**

```
1  int main() {  
2  
3      int num = 0;  
4  
5      do  
6      {  
7          cout << num << endl;  
8          num++;  
9  
10     } while (num < 10);  
11  
12  
13     system("pause");  
14  
15     return 0;  
16 }
```

总结：与while循环区别在于，do...while先执行一次循环语句，再判断循环条件

### 练习案例：水仙花数

**案例描述：**水仙花数是指一个 3 位数，它的每个位上的数字的 3 次幂之和等于它本身

例如： $1^3 + 5^3 + 3^3 = 153$

请利用do...while语句，求出所有3位数中的水仙花数

### 4.2.3 for循环语句

**作用：** 满足循环条件，执行循环语句

**语法：** `for(起始表达式;条件表达式;末尾循环体) { 循环语句; }`

**示例：**

```
1  int main() {  
2  
3      for (int i = 0; i < 10; i++)  
4      {  
5          cout << i << endl;  
6      }  
7  
8      system("pause");  
9  
10     return 0;  
11 }
```

**详解：**

1541673704101

注意：for循环中的表达式，要用分号进行分隔

总结：while , do...while, for都是开发中常用的循环语句，for循环结构比较清晰，比较常用

#### 练习案例：敲桌子

案例描述：从1开始数到数字100，如果数字个位含有7，或者数字十位含有7，或者该数字是7的倍数，我们打印敲桌子，其余数字直接打印输出。

timg

## 4.2.4 嵌套循环

**作用：** 在循环体中再嵌套一层循环，解决一些实际问题

例如我们想在屏幕中打印如下图片，就需要利用嵌套循环



**示例：**

```
1  int main() {
2
3      //外层循环执行1次，内层循环执行1轮
4      for (int i = 0; i < 10; i++)
5      {
6          for (int j = 0; j < 10; j++)
7          {
8              cout << "*" << " ";
9          }
10         cout << endl;
11     }
12
13     system("pause");
14
15     return 0;
16 }
```

**练习案例：** 乘法口诀表

案例描述：利用嵌套循环，实现九九乘法表



## 4.3 跳转语句

### 4.3.1 break语句

**作用:** 用于跳出选择结构或者循环结构

break使用的时机:

- 出现在switch条件语句中，作用是终止case并跳出switch
- 出现在循环语句中，作用是跳出当前的循环语句
- 出现在嵌套循环中，跳出最近的内层循环语句

**示例1:**

```
1  int main() {
2      //1、在switch 语句中使用break
3      cout << "请选择您挑战副本的难度: " << endl;
4      cout << "1、普通" << endl;
5      cout << "2、中等" << endl;
6      cout << "3、困难" << endl;
7
8      int num = 0;
9
10     cin >> num;
11
12     switch (num)
13     {
14     case 1:
15         cout << "您选择的是普通难度" << endl;
16         break;
17     case 2:
18         cout << "您选择的是中等难度" << endl;
19         break;
20     case 3:
21         cout << "您选择的是困难难度" << endl;
22         break;
23     }
24
25     system("pause");
26
27     return 0;
28 }
```

**示例2:**

```
1  int main() {
2      //2、在循环语句中用break
3      for (int i = 0; i < 10; i++)
4      {
5          if (i == 5)
6          {
7              break; //跳出循环语句
8          }
9          cout << i << endl;
10     }
```

```

10     }
11
12     system("pause");
13
14     return 0;
15 }

```

### 示例3:

```

1  int main() {
2      //在嵌套循环语句中使用break, 退出内层循环
3      for (int i = 0; i < 10; i++)
4      {
5          for (int j = 0; j < 10; j++)
6          {
7              if (j == 5)
8              {
9                  break;
10             }
11             cout << "*" << " ";
12         }
13         cout << endl;
14     }
15
16     system("pause");
17
18     return 0;
19 }

```

## 4.3.2 continue语句

**作用:** 在循环语句中, 跳过本次循环中余下尚未执行的语句, 继续执行下一次循环

**示例:**

```

1  int main() {
2
3      for (int i = 0; i < 100; i++)
4      {
5          if (i % 2 == 0)
6          {
7              continue;
8          }

```

```

9         cout << i << endl;
10     }
11
12     system("pause");
13
14     return 0;
15 }

```

注意：continue并没有使整个循环终止，而break会跳出循环

### 4.3.3 goto语句

**作用：**可以无条件跳转语句

**语法：**goto 标记;

**解释：**如果标记的名称存在，执行到goto语句时，会跳转到标记的位置

**示例：**

```

1  int main() {
2
3      cout << "1" << endl;
4
5      goto FLAG;
6
7      cout << "2" << endl;
8      cout << "3" << endl;
9      cout << "4" << endl;
10
11     FLAG:
12
13     cout << "5" << endl;
14
15     system("pause");
16
17     return 0;
18 }

```

注意：在程序中不建议使用goto语句，以免造成程序流程混乱



## 5 数组

---

### 5.1 概述

所谓数组，就是一个集合，里面存放了相同类型的数据元素

**特点1：**数组中的每个数据元素都是相同的数据类型

**特点2：**数组是由连续的内存位置组成的



1541748375356

### 5.2 一维数组

#### 5.2.1 一维数组定义方式

一维数组定义的三种方式：

1. 数据类型 数组名[ 数组长度 ];
2. 数据类型 数组名[ 数组长度 ] = { 值1, 值2 ...};
3. 数据类型 数组名[ ] = { 值1, 值2 ...};

示例

```
1 int main() {  
2
```

```

3      //定义方式1
4      //数据类型 数组名[元素个数];
5      int score[10];
6
7      //利用下标赋值
8      score[0] = 100;
9      score[1] = 99;
10     score[2] = 85;
11
12     //利用下标输出
13     cout << score[0] << endl;
14     cout << score[1] << endl;
15     cout << score[2] << endl;
16
17
18     //第二种定义方式
19     //数据类型 数组名[元素个数] = {值1, 值2 , 值3 ...};
20     //如果{}内不足10个数据，剩余数据用0补全
21     int score2[10] = { 100, 90,80,70,60,50,40,30,20,10 };
22
23     //逐个输出
24     //cout << score2[0] << endl;
25     //cout << score2[1] << endl;
26
27     //一个一个输出太麻烦，因此可以利用循环进行输出
28     for (int i = 0; i < 10; i++)
29     {
30         cout << score2[i] << endl;
31     }
32
33     //定义方式3
34     //数据类型 数组名[] = {值1, 值2 , 值3 ...};
35     int score3[] = { 100,90,80,70,60,50,40,30,20,10 };
36
37     for (int i = 0; i < 10; i++)
38     {
39         cout << score3[i] << endl;
40     }
41
42     system("pause");
43
44     return 0;
45 }

```

总结1：数组名的命名规范与变量名命名规范一致，不要和变量重名

总结2：数组中下标是从0开始索引

## 5.2.2 一维数组数组名

一维数组名称的用途：

1. 可以统计整个数组在内存中的长度
2. 可以获取数组在内存中的首地址

示例：

```
1  int main() {
2
3      //数组名用途
4      //1、可以获取整个数组占用内存空间大小
5      int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
6
7      cout << "整个数组所占内存空间为: " << sizeof(arr) << endl;
8      cout << "每个元素所占内存空间为: " << sizeof(arr[0]) << endl;
9      cout << "数组的元素个数为: " << sizeof(arr) / sizeof(arr[0]) << endl;
10
11     //2、可以通过数组名获取到数组首地址
12     cout << "数组首地址为: " << (int)arr << endl;
13     cout << "数组中第一个元素地址为: " << (int)&arr[0] << endl;
14     cout << "数组中第二个元素地址为: " << (int)&arr[1] << endl;
15
16     //arr = 100; 错误，数组名是常量，因此不可以赋值
17
18
19     system("pause");
20
21     return 0;
22 }
```

注意：数组名是常量，不可以赋值

总结1：直接打印数组名，可以查看数组所占内存的首地址

总结2：对数组名进行sizeof，可以获取整个数组占内存空间的大小

### 练习案例1：五只小猪称体重

案例描述：

在一个数组中记录了五只小猪的体重，如：int arr[5] = {300,350,200,400,250};

找出并打印最重的小猪体重。

### 练习案例2：数组元素逆置


**案例描述：**请声明一个5个元素的数组，并且将元素逆置。

(如原数组元素为：1,3,2,5,4;逆置后输出结果为:4,5,2,3,1);

## 5.2.3 冒泡排序

**作用：**最常用的排序算法，对数组内元素进行排序

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素做同样的工作，执行完毕后，找到第一个最大值。
3. 重复以上的步骤，每次比较次数-1，直到不需要比较

1541905327273

**示例：**将数组 { 4,2,8,0,5,7,1,3,9 } 进行升序排序

```
1  int main() {
2
3      int arr[9] = { 4,2,8,0,5,7,1,3,9 };
4
5      for (int i = 0; i < 9 - 1; i++)
6      {
7          for (int j = 0; j < 9 - 1 - i; j++)
8          {
9              if (arr[j] > arr[j + 1])
10             {
11                 int temp = arr[j];
12                 arr[j] = arr[j + 1];
13                 arr[j + 1] = temp;
14             }
15         }
16     }
17
18     for (int i = 0; i < 9; i++)
19     {
20         cout << arr[i] << endl;
```


```

21     }
22
23     system("pause");
24
25     return 0;
26 }

```

## 5.3 二维数组

二维数组就是在一维数组上，多加一个维度。

 1541905559138

### 5.3.1 二维数组定义方式

二维数组定义的四种方式：

1. 数据类型 数组名 [ 行数 ] [ 列数 ] ;
2. 数据类型 数组名 [ 行数 ] [ 列数 ] = { {数据1, 数据2 } , {数据3, 数据4 } } ;
3. 数据类型 数组名 [ 行数 ] [ 列数 ] = { 数据1, 数据2, 数据3, 数据4 } ;
4. 数据类型 数组名 [ ] [ 列数 ] = { 数据1, 数据2, 数据3, 数据4 } ;

建议：以上4种定义方式，利用**第二种更加直观，提高代码的可读性**

示例：

```

1  int main() {
2
3      //方式1
4      //数组类型 数组名 [行数][列数]
5      int arr[2][3];
6      arr[0][0] = 1;
7      arr[0][1] = 2;
8      arr[0][2] = 3;
9      arr[1][0] = 4;
10     arr[1][1] = 5;
11     arr[1][2] = 6;
12
13     for (int i = 0; i < 2; i++)
14     {
15         for (int j = 0; j < 3; j++)
16         {
17             cout << arr[i][j] << " ";
18         }
19         cout << endl;
20     }
21
22     //方式2
23     //数据类型 数组名[行数][列数] = { {数据1, 数据2 } , {数据3, 数据4 } } ;

```

```

24     int arr2[2][3] =
25     {
26         {1,2,3},
27         {4,5,6}
28     };
29
30     //方式3
31     //数据类型 数组名[行数][列数] = { 数据1, 数据2 ,数据3, 数据4  };
32     int arr3[2][3] = { 1,2,3,4,5,6 };
33
34     //方式4
35     //数据类型 数组名[][列数] = { 数据1, 数据2 ,数据3, 数据4  };
36     int arr4[][3] = { 1,2,3,4,5,6 };
37
38     system("pause");
39
40     return 0;
41 }

```

总结：在定义二维数组时，如果初始化了数据，可以省略行数

### 5.3.2 二维数组数组名

- 查看二维数组所占内存空间
- 获取二维数组首地址

示例：

```

1  int main() {
2
3      //二维数组数组名
4      int arr[2][3] =
5      {
6          {1,2,3},
7          {4,5,6}
8      };
9
10     cout << "二维数组大小: " << sizeof(arr) << endl;
11     cout << "二维数组一行大小: " << sizeof(arr[0]) << endl;
12     cout << "二维数组元素大小: " << sizeof(arr[0][0]) << endl;

```

```

13
14     cout << "二维数组行数: " << sizeof(arr) / sizeof(arr[0]) << endl;
15     cout << "二维数组列数: " << sizeof(arr[0]) / sizeof(arr[0][0]) << endl;
16
17     //地址
18     cout << "二维数组首地址: " << arr << endl;
19     cout << "二维数组第一行地址: " << arr[0] << endl;
20     cout << "二维数组第二行地址: " << arr[1] << endl;
21
22     cout << "二维数组第一个元素地址: " << &arr[0][0] << endl;
23     cout << "二维数组第二个元素地址: " << &arr[0][1] << endl;
24
25     system("pause");
26
27     return 0;
28 }

```

总结1: 二维数组名就是这个数组的首地址

总结2: 对二维数组名进行sizeof时, 可以获取整个二维数组占用的内存空间大小

### 5.3.3 二维数组应用案例

#### 考试成绩统计:

案例描述: 有三名同学 (张三, 李四, 王五), 在一次考试中的成绩分别如下表, 请分别输出三名同学的总成绩

	语文	数学	英语
张三	100	100	100
李四	90	50	100
王五	60	70	80

#### 参考答案:

```

1  int main() {
2
3      int scores[3][3] =
4      {

```

```

5         {100,100,100},
6         {90,50,100},
7         {60,70,80},
8     };
9
10    string names[3] = { "张三","李四","王五" };
11
12    for (int i = 0; i < 3; i++)
13    {
14        int sum = 0;
15        for (int j = 0; j < 3; j++)
16        {
17            sum += scores[i][j];
18        }
19        cout << names[i] << "同学总成绩为: " << sum << endl;
20    }
21
22    system("pause");
23
24    return 0;
25 }

```

## 6 函数

### 6.1 概述

**作用：**将一段经常使用的代码封装起来，减少重复代码

一个较大的程序，一般分为若干个程序块，每个模块实现特定的功能。

### 6.2 函数的定义

函数的定义一般主要有5个步骤：

- 1、返回值类型
- 2、函数名
- 3、参数表列
- 4、函数体语句
- 5、return 表达式

**语法：**



```
1 返回值类型 函数名 （参数列表）
2  {
3
4      函数体语句
5
6      return表达式
7
8  }
```

- 返回值类型：一个函数可以返回一个值。在函数定义中
- 函数名：给函数起个名称
- 参数列表：使用该函数时，传入的数据
- 函数体语句：花括号内的代码，函数内需要执行的语句
- return表达式：和返回值类型挂钩，函数执行完后，返回相应的数据

**示例：**定义一个加法函数，实现两个数相加

```
1  //函数定义
2  int add(int num1, int num2)
3  {
4      int sum = num1 + num2;
5      return sum;
6  }
```

## 6.3 函数的调用

**功能：**使用定义好的函数

**语法：** 函数名（参数）

**示例：**

```
1  //函数定义
2  int add(int num1, int num2) //定义中的num1,num2称为形式参数，简称形参
3  {
4      int sum = num1 + num2;
5      return sum;
6  }
7
8  int main() {
9
10     int a = 10;
```

```

11     int b = 10;
12     //调用add函数
13     int sum = add(a, b); //调用时的a, b称为实际参数, 简称实参
14     cout << "sum = " << sum << endl;
15
16     a = 100;
17     b = 100;
18
19     sum = add(a, b);
20     cout << "sum = " << sum << endl;
21
22     system("pause");
23
24     return 0;
25 }

```

总结：函数定义里小括号内称为形参，函数调用时传入的参数称为实参

## 6.4 值传递

- 所谓值传递，就是函数调用时实参将数值传入给形参
- 值传递时，如果形参发生，并不会影响实参

示例：

```

1 void swap(int num1, int num2)
2 {
3     cout << "交换前: " << endl;
4     cout << "num1 = " << num1 << endl;
5     cout << "num2 = " << num2 << endl;
6
7     int temp = num1;
8     num1 = num2;
9     num2 = temp;
10
11     cout << "交换后: " << endl;
12     cout << "num1 = " << num1 << endl;
13     cout << "num2 = " << num2 << endl;
14
15     //return ; 当函数声明时候, 不需要返回值, 可以不写return
16 }
17
18 int main() {
19
20     int a = 10;
21     int b = 20;
22
23     swap(a, b);
24

```

```

25     cout << "mian中的 a = " << a << endl;
26     cout << "mian中的 b = " << b << endl;
27
28     system("pause");
29
30     return 0;
31 }

```

总结：值传递时，形参是修饰不了实参的

## 6.5 函数的常见样式

常见的函数样式有4种

1. 无参无返
2. 有参无返
3. 无参有返
4. 有参有返

示例：

```

1 //函数常见样式
2 //1、 无参无返
3 void test01()
4 {
5     //void a = 10; //无类型不可以创建变量,原因无法分配内存
6     cout << "this is test01" << endl;
7     //test01(); 函数调用
8 }
9
10 //2、 有参无返
11 void test02(int a)
12 {
13     cout << "this is test02" << endl;
14     cout << "a = " << a << endl;
15 }
16
17 //3、 无参有返
18 int test03()
19 {
20     cout << "this is test03 " << endl;
21     return 10;
22 }
23
24 //4、 有参有返
25 int test04(int a, int b)
26 {
27     cout << "this is test04 " << endl;
28     int sum = a + b;

```

```
29     return sum;
30 }
```

## 6.6 函数的声明

**作用：** 告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

- 函数的**声明**可以多次，但是函数的**定义**只能有一次

**示例：**

```
1  //声明可以多次，定义只能一次
2  //声明
3  int max(int a, int b);
4  int max(int a, int b);
5  //定义
6  int max(int a, int b)
7  {
8      return a > b ? a : b;
9  }
10
11 int main() {
12
13     int a = 100;
14     int b = 200;
15
16     cout << max(a, b) << endl;
17
18     system("pause");
19
20     return 0;
21 }
```

## 6.7 函数的分文件编写

**作用：**让代码结构更加清晰

函数分文件编写一般有4个步骤

1. 创建后缀名为.h的头文件
2. 创建后缀名为.cpp的源文件
3. 在头文件中写函数的声明
4. 在源文件中写函数的定义

**示例：**

```
1 //swap.h文件
2 #include<iostream>
3 using namespace std;
4
5 //实现两个数字交换的函数声明
6 void swap(int a, int b);
7
```

```
1 //swap.cpp文件
2 #include "swap.h"
3
4 void swap(int a, int b)
5 {
6     int temp = a;
7     a = b;
8     b = temp;
9
10    cout << "a = " << a << endl;
11    cout << "b = " << b << endl;
12 }
```

```
1 //main函数文件
2 #include "swap.h"
3 int main() {
4
5     int a = 100;
6     int b = 200;
7     swap(a, b);
8
9     system("pause");
10
11    return 0;
12 }
13
```

# 7 指针

## 7.1 指针的基本概念

**指针的作用：** 可以通过指针间接访问内存

- 内存编号是从0开始记录的，一般用十六进制数字表示
- 可以利用指针变量保存地址

## 7.2 指针变量的定义和使用

指针变量定义语法： `数据类型 * 变量名；`

**示例：**

```
1  int main() {
2
3      //1、指针的定义
4      int a = 10; //定义整型变量a
5
6      //指针定义语法： 数据类型 * 变量名 ；
7      int * p;
8
9      //指针变量赋值
10     p = &a; //指针指向变量a的地址
11     cout << &a << endl; //打印数据a的地址
12     cout << p << endl;  //打印指针变量p
13
14     //2、指针的使用
15     //通过*操作指针变量指向的内存
16     cout << "*p = " << *p << endl;
17
18     system("pause");
19
20     return 0;
21 }
```

**指针变量和普通变量的区别**

- 普通变量存放的是数据,指针变量存放的是地址
- 指针变量可以通过" \* "操作符，操作指针变量指向的内存空间，这个过程称为解引用

总结1： 我们可以通过 & 符号 获取变量的地址

总结2： 利用指针可以记录地址

总结3： 对指针变量解引用，可以操作指针指向的内存

## 7.3 指针所占内存空间

提问：指针也是种数据类型，那么这种数据类型占用多少内存空间？

示例：

```
1  int main() {
2
3      int a = 10;
4
5      int * p;
6      p = &a; //指针指向数据a的地址
7
8      cout << *p << endl; /* 解引用
9      cout << sizeof(p) << endl;
10     cout << sizeof(char *) << endl;
11     cout << sizeof(float *) << endl;
12     cout << sizeof(double *) << endl;
13
14     system("pause");
15
16     return 0;
17 }
```

总结：所有指针类型在32位操作系统下是4个字节，不管是什么类型的指针都一样  
在64位操作系统下指针占8个字节

## 7.4 空指针和野指针

**空指针：**指针变量指向内存中编号为0的空间

**用途：**初始化指针变量

**注意：**空指针指向的内存是不可以访问的；内存编号0 ~ 255为系统占用内存，不允许用户访问

## 示例1: 空指针

```
1  int main() {
2
3      //指针变量p指向内存地址编号为0的空间
4      int * p = NULL;
5
6      //访问空指针报错
7      //内存编号0 ~255为系统占用内存，不允许用户访问
8      cout << *p << endl;
9
10     system("pause");
11
12     return 0;
13 }
```

**野指针:** 指针变量指向非法的内存空间

## 示例2: 野指针

```
1  int main() {
2
3      //指针变量p指向内存地址编号为0x1100的空间
4      int * p = (int *)0x1100;
5
6      //访问野指针报错
7      cout << *p << endl;
8
9      system("pause");
10
11     return 0;
12 }
```

总结: 空指针和野指针都不是我们申请的空间, 因此不要访问。



## 7.5 const修饰指针

const修饰指针有三种情况

1. const修饰指针 --- 常量指针
2. const修饰常量 --- 指针常量
3. const即修饰指针，又修饰常量

示例：

```
1  int main() {
2
3      int a = 10;
4      int b = 10;
5
6      //const修饰的是指针，指针指向可以改，指针指向的值不可以更改
7      const int * p1 = &a;
8      p1 = &b; //正确
9      /*p1 = 100; 报错
10
11
12     //const修饰的是常量，指针指向不可以改，指针指向的值可以更改
13     int * const p2 = &a;
14     //p2 = &b; //错误
15     *p2 = 100; //正确
16
17     //const既修饰指针又修饰常量
18     const int * const p3 = &a;
19     //p3 = &b; //错误
20     //*p3 = 100; //错误
21
22     system("pause");
23
24     return 0;
25 }
```

技巧：看const右侧紧跟着的是指针还是常量，是指针就是常量指针，是常量就是指针常量

## 7.6 指针和数组

作用：利用指针访问数组中元素

示例：

```
1  int main() {
2
3      int arr[] = { 1,2,3,4,5,6,7,8,9,10 };
```

```

4
5     int * p = arr; //指向数组的指针
6
7     cout << "第一个元素:  " << arr[0] << endl;
8     cout << "指针访问第一个元素:  " << *p << endl;
9
10    for (int i = 0; i < 10; i++)
11    {
12        //利用指针遍历数组
13        cout << *p << endl;
14        p++;
15    }
16
17    system("pause");
18
19    return 0;
20 }

```

## 7.7 指针和函数

**作用：**利用指针作函数参数，可以修改实参的值

**示例：**

```

1 //值传递
2 void swap1(int a ,int b)
3 {
4     int temp = a;
5     a = b;
6     b = temp;
7 }
8 //地址传递
9 void swap2(int * p1, int *p2)
10 {
11     int temp = *p1;
12     *p1 = *p2;
13     *p2 = temp;
14 }
15
16 int main() {
17
18     int a = 10;
19     int b = 20;
20     swap1(a, b); // 值传递不会改变实参
21
22     swap2(&a, &b); //地址传递会改变实参
23

```

```

24     cout << "a = " << a << endl;
25
26     cout << "b = " << b << endl;
27
28     system("pause");
29
30     return 0;
31 }

```

总结：如果不想修改实参，就用值传递，如果想修改实参，就用地址传递

## 7.8 指针、数组、函数

**案例描述：**封装一个函数，利用冒泡排序，实现对整型数组的升序排序

例如数组：int arr[10] = { 4,3,6,9,1,2,10,8,7,5};

**示例：**

```

1  //冒泡排序函数
2  void bubblesort(int * arr, int len) //int * arr 也可以写为int arr[]
3  {
4      for (int i = 0; i < len - 1; i++)
5      {
6          for (int j = 0; j < len - 1 - i; j++)
7          {
8              if (arr[j] > arr[j + 1])
9              {
10                 int temp = arr[j];
11                 arr[j] = arr[j + 1];
12                 arr[j + 1] = temp;
13             }
14         }
15     }
16 }
17
18 //打印数组函数
19 void printArray(int arr[], int len)
20 {
21     for (int i = 0; i < len; i++)
22     {
23         cout << arr[i] << endl;
24     }
25 }

```

```

26
27 int main() {
28
29     int arr[10] = { 4,3,6,9,1,2,10,8,7,5 };
30     int len = sizeof(arr) / sizeof(int);
31
32     bubbleSort(arr, len);
33
34     printArray(arr, len);
35
36     system("pause");
37
38     return 0;
39 }

```

总结：当数组名传入到函数作为参数时，被退化为指向首元素的指针

## 8 结构体

### 8.1 结构体基本概念

结构体属于用户自定义的数据类型，允许用户存储不同的数据类型

### 8.2 结构体定义和使用

**语法：** `struct 结构体名 { 结构体成员列表 };`

通过结构体创建变量的方式有三种：

- `struct 结构体名 变量名`
- `struct 结构体名 变量名 = { 成员1值, 成员2值...}`
- 定义结构体时顺便创建变量

**示例：**

```

1 //结构体定义
2 struct student
3 {
4     //成员列表
5     string name; //姓名
6     int age;     //年龄
7     int score;   //分数
8 }stu3; //结构体变量创建方式3
9
10
11 int main() {
12

```

```

13 //结构体变量创建方式1
14 struct student stu1; //struct 关键字可以省略
15
16 stu1.name = "张三";
17 stu1.age = 18;
18 stu1.score = 100;
19
20 cout << "姓名: " << stu1.name << " 年龄: " << stu1.age << " 分数: " <<
stu1.score << endl;
21
22 //结构体变量创建方式2
23 struct student stu2 = { "李四", 19, 60 };
24
25 cout << "姓名: " << stu2.name << " 年龄: " << stu2.age << " 分数: " <<
stu2.score << endl;
26
27
28 stu3.name = "王五";
29 stu3.age = 18;
30 stu3.score = 80;
31
32
33 cout << "姓名: " << stu3.name << " 年龄: " << stu3.age << " 分数: " <<
stu3.score << endl;
34
35 system("pause");
36
37 return 0;
38 }

```

总结1: 定义结构体时的关键字是struct, 不可省略

总结2: 创建结构体变量时, 关键字struct可以省略

总结3: 结构体变量利用操作符 "." 访问成员

## 8.3 结构体数组

**作用:** 将自定义的结构体放入到数组中方便维护

**语法:** `struct 结构体名 数组名[元素个数] = { {}, {}, ... {} }`

**示例:**

```

1 //结构体定义
2 struct student
3 {
4     //成员列表
5     string name; //姓名
6     int age; //年龄

```

```

7      int score;    //分数
8  }
9
10 int main() {
11
12     //结构体数组
13     struct student arr[3]=
14     {
15         {"张三",18,80 },
16         {"李四",19,60 },
17         {"王五",20,70 }
18     };
19
20     for (int i = 0; i < 3; i++)
21     {
22         cout << "姓名: " << arr[i].name << " 年龄: " << arr[i].age << " 分数: "
23         << arr[i].score << endl;
24     }
25
26     system("pause");
27
28     return 0;
29 }

```

## 8.4 结构体指针

**作用：**通过指针访问结构体中的成员

- 利用操作符 `->` 可以通过结构体指针访问结构体属性

**示例：**

```

1  //结构体定义
2  struct student
3  {
4      //成员列表
5      string name; //姓名
6      int age;     //年龄
7      int score;   //分数
8  };
9
10
11 int main() {
12
13     struct student stu = { "张三",18,100, };

```

```

14
15     struct student * p = &stu;
16
17     p->score = 80; //指针通过 -> 操作符可以访问成员
18
19     cout << "姓名: " << p->name << " 年龄: " << p->age << " 分数: " << p->score
    << endl;
20
21     system("pause");
22
23     return 0;
24 }

```

总结：结构体指针可以通过 -> 操作符 来访问结构体中的成员

## 8.5 结构体嵌套结构体

**作用：** 结构体中的成员可以是另一个结构体

**例如：** 每个老师辅导一个学员，一个老师的结构体中，记录一个学生的结构体

**示例：**

```

1  //学生结构体定义
2  struct student
3  {
4      //成员列表
5      string name; //姓名
6      int age; //年龄
7      int score; //分数
8  };
9
10 //教师结构体定义
11 struct teacher
12 {
13     //成员列表
14     int id; //职工编号
15     string name; //教师姓名
16     int age; //教师年龄
17     struct student stu; //子结构体 学生
18 };
19
20
21 int main() {
22
23     struct teacher t1;

```

```

24     t1.id = 10000;
25     t1.name = "老王";
26     t1.age = 40;
27
28     t1.stu.name = "张三";
29     t1.stu.age = 18;
30     t1.stu.score = 100;
31
32     cout << "教师 职工编号: " << t1.id << " 姓名: " << t1.name << " 年龄: "
    << t1.age << endl;
33
34     cout << "辅导学员 姓名: " << t1.stu.name << " 年龄: " << t1.stu.age << " 考
    试分数: " << t1.stu.score << endl;
35
36     system("pause");
37
38     return 0;
39 }

```

**总结：**在结构体中可以定义另一个结构体作为成员，用来解决实际问题

## 8.6 结构体做函数参数

**作用：**将结构体作为参数向函数中传递

传递方式有两种：

- 值传递
- 地址传递

**示例：**

```

1  //学生结构体定义
2  struct student
3  {
4      //成员列表
5      string name; //姓名
6      int age; //年龄
7      int score; //分数
8  };
9
10 //值传递
11 void printStudent(student stu )
12 {
13     stu.age = 28;
14     cout << "子函数中 姓名: " << stu.name << " 年龄: " << stu.age << " 分数: "
    << stu.score << endl;
15 }
16

```



```

17 //地址传递
18 void printStudent2(student *stu)
19 {
20     stu->age = 28;
21     cout << "子函数中 姓名: " << stu->name << " 年龄: " << stu->age << " 分
数: " << stu->score << endl;
22 }
23
24 int main() {
25
26     student stu = { "张三",18,100};
27     //值传递
28     printStudent(stu);
29     cout << "主函数中 姓名: " << stu.name << " 年龄: " << stu.age << " 分数: "
<< stu.score << endl;
30
31     cout << endl;
32
33     //地址传递
34     printStudent2(&stu);
35     cout << "主函数中 姓名: " << stu.name << " 年龄: " << stu.age << " 分数: "
<< stu.score << endl;
36
37     system("pause");
38
39     return 0;
40 }

```

总结: 如果不想修改主函数中的数据, 用值传递, 反之用地址传递

## 8.7 结构体中 const使用场景

**作用:** 用const来防止误操作

**示例:**

```

1 //学生结构体定义
2 struct student
3 {
4     //成员列表
5     string name; //姓名
6     int age; //年龄
7     int score; //分数
8 };
9
10 //const使用场景
11 void printStudent(const student *stu) //加const防止函数体中的误操作
12 {
13     //stu->age = 100; //操作失败, 因为加了const修饰
14     cout << "姓名: " << stu->name << " 年龄: " << stu->age << " 分数: " << stu-
>score << endl;
15

```

```

16 }
17
18 int main() {
19
20     student stu = { "张三", 18, 100 };
21
22     printStudent(&stu);
23
24     system("pause");
25
26     return 0;
27 }

```

## 8.8 结构体案例

### 8.8.1 案例1

#### 案例描述：

学校正在做毕设项目，每名老师带领5个学生，总共有3名老师，需求如下

设计学生和老师的结构体，其中在老师的结构体中，有老师姓名和一个存放5名学生的数组作为成员

学生的成员有姓名、考试分数，创建数组存放3名老师，通过函数给每个老师及所带的学生赋值

最终打印出老师数据以及老师所带的学生数据。

#### 示例：

```

1  struct Student
2  {
3      string name;
4      int score;
5  };
6  struct Teacher
7  {
8      string name;
9      Student sArray[5];
10 };
11
12 void allocateSpace(Teacher tArray[] , int len)
13 {
14     string tName = "教师";
15     string sName = "学生";
16     string nameSeed = "ABCDE";
17     for (int i = 0; i < len; i++)
18     {
19         tArray[i].name = tName + nameSeed[i];
20
21         for (int j = 0; j < 5; j++)

```

```

22     {
23         tArray[i].sArray[j].name = sName + nameSeed[j];
24         tArray[i].sArray[j].score = rand() % 61 + 40;
25     }
26 }
27 }
28
29 void printTeachers(Teacher tArray[], int len)
30 {
31     for (int i = 0; i < len; i++)
32     {
33         cout << tArray[i].name << endl;
34         for (int j = 0; j < 5; j++)
35         {
36             cout << "\t姓名: " << tArray[i].sArray[j].name << " 分数: " <<
tArray[i].sArray[j].score << endl;
37         }
38     }
39 }
40
41 int main() {
42
43     srand((unsigned int)time(NULL)); //随机数种子 头文件 #include <ctime>
44
45     Teacher tArray[3]; //老师数组
46
47     int len = sizeof(tArray) / sizeof(Teacher);
48
49     allocatespace(tArray, len); //创建数据
50
51     printTeachers(tArray, len); //打印数据
52
53     system("pause");
54
55     return 0;
56 }

```

## 8.8.2 案例2

### 案例描述:

设计一个英雄的结构体，包括成员姓名，年龄，性别;创建结构体数组，数组中存放5名英雄。

通过冒泡排序的算法，将数组中的英雄按照年龄进行升序排序，最终打印排序后的结果。

五名英雄信息如下:

```

1      {"刘备",23,"男"},
2      {"关羽",22,"男"},
3      {"张飞",20,"男"},
4      {"赵云",21,"男"},
5      {"貂蝉",19,"女"},

```

**示例:**

```

1  //英雄结构体
2  struct hero
3  {
4      string name;
5      int age;
6      string sex;
7  };
8  //冒泡排序
9  void bubblesort(hero arr[] , int len)
10 {
11     for (int i = 0; i < len - 1; i++)
12     {
13         for (int j = 0; j < len - 1 - i; j++)
14         {
15             if (arr[j].age > arr[j + 1].age)
16             {
17                 hero temp = arr[j];
18                 arr[j] = arr[j + 1];
19                 arr[j + 1] = temp;
20             }
21         }
22     }
23 }
24 //打印数组
25 void printHeros(hero arr[], int len)
26 {
27     for (int i = 0; i < len; i++)
28     {
29         cout << "姓名: " << arr[i].name << " 性别: " << arr[i].sex << " 年
年龄: " << arr[i].age << endl;
30     }
31 }
32
33 int main() {
34
35     struct hero arr[5] =
36     {
37         {"刘备",23,"男"},
38         {"关羽",22,"男"},
39         {"张飞",20,"男"},
40         {"赵云",21,"男"},
41         {"貂蝉",19,"女"},

```

```

42     };
43
44     int len = sizeof(arr) / sizeof(hero); //获取数组元素个数
45
46     bubbleSort(arr, len); //排序
47
48     printHeros(arr, len); //打印
49
50     system("pause");
51
52     return 0;
53 }

```

- 构造函数：函数名一定要和类名相同，构造函数有初始列，可以进行赋值

```

1 public:
2     complex (double r = 0, double i = 0) //构造函数，默认实参r=0,i=0
3         : re (r), im (i) //利用构造函数的特性进行赋值（初值列）
4     { }

```

在大括号中初始化会导致效率降低

- 不带指针的类大部分不会使用构造函数
- 构造函数可以有很多个--**重载**
- 函数重载经常发生在构造函数中
- 构造函数可以放在private中（singleton）
- 一个类中可以有多public和private
- 常量函数const(不改变函数中数据的内容)

```

1 double real() const { return re; }

```

- 常量函数调用时参数也需要是常量

```

1 class complex
2 {
3     public:
4     double real() const { return re; }
5     double imag() const { return im; }
6 }
7 ...
8     const complex c1(2,1)

```

- 参数传递（在实际编程中最好不要直接传递值，传递reference即可）
- 返回值传递（会影响程序的运行效率）
- **友元函数**：friend：定义在私有类但是外部函数也可以调用
- 同一个类中的各个对象互为友元函数
- 返回值最好用reference来传递

- 内存分区：

- 代码区：特点：共享，只读
- 全局区：存放全局变量，
  - 静态变量（static），
  - 常量
    - 字符串常量
    - const修饰常量（const修饰的全局变量和局部变量）
- 堆区：由程序员自己管理释放，利用new关键字在堆区开辟数据区域
- 栈区：由编译器自动分配，存放函数的参数值，局部变量**由于是编译器自动释放，不要返回局部变量的地址**
- new操作符：利用new操作符在堆区开辟数据区域，由程序员手动开辟，手动释放（利用delete释放）
  - new返回的是该数据类型的指针

## 引用

- 引用：给变量取别名

- ```
1 | int &b = a; //b为别名
```

- 引用必须要初始化；引用初始化之后不可以再发生改变
- 如果函数的返回值是引用，则该函数调用可以作为左值

- ```
1 | int& swap()
2 | {
3 |     static int a = 10; //静态变量在全局区，程序结束后系统自动释放
4 |     return a;
5 | }
6 | swap() = 1000;
```

- 引用的本质就是**指针常量**（地址不变值可以变）
- 常量引用：主要用来修饰形参，防止误操作

- ```
1 | const int& ref = 10;
2 | void show(const int& val) //修饰形参，在函数体内不可再次修改
3 | {
4 |     cout << val;
5 | }
```

- 函数默认参数：可以给形参设置默认值

```
1 | int func(int a, int b = 100)
2 | {
3 |     return a+b;
4 | }
5 | func(a) //有默认形参的函数可以少传参数值
```

- 如果某个位置有了默认参数，则从这个位置开始从左到右都必须有默认参数
- 如果函数的声明有默认参数，函数的实现就不能有默认参数了（声明和实现只能有一个有默认参数，否则会出现二义性）
- 占位参数

```
1 void func(int a, int);
2 void func(int a, int 10)//占位参数可以有默认参数
3 func(10, 10);//必须要有，但是不能用
```

- 函数重载：函数名相同，提高复用性
  - 满足条件：同一作用域下；函数名相同；函数参数类型不同，或者个数不同，顺序不同
  - 引用作为函数重载
  - 重载遇到参数默认值
- 类中的属性和行为同一称为成员，属性也称为成员属性和成员变量；行为称为成员函数和成员方法
- 封装的含义：
  - public公共权限：在类内和类外都可以访问
  - protected保护：在类内可以访问，类外不可以访问
  - private私有权限：在类内可以访问，类外不可以访问（在继承时保护和私有权限有区别）
- struct和class唯一的区别在于默认访问权限不同（struct默认为公有，class默认权限为私有）
- 成员属性设为私有后可以自己控制读写权限

## 构造函数和析构函数

- 利用**构造函数**和**析构函数**对对象进行初始化和清理，这两个函数可以由编译器自动调用，编译器提供的构造函数和析构函数是空的
- 构造函数：用来初始化，可以有参数，可以发生重载，函数名称与类名相同，没有返回值也不用写void

- 1 | 类名 () {}

- 构造函数的分类：
  - 有参构造和无参构造（默认构造函数）
  - 普通构造函数和拷贝构造函数
- 构造函数的调用：
  - 括号法
  - 显示法
  - 隐式转换法
- 匿名对象：当前行执行结束后，系统会立即回收掉匿名对象；**不要利用拷贝构造函数来初始化匿名对象**，编译器会认为是个对象的声明
- 拷贝构造函数的使用时机：
  - 使用一个已经创建完毕的对象来初始化一个新对象
  - 值传递的方式给函数参数传值
  - 以值方式返回局部对象
- 如果用户定义有参构造函数，c++不再提供默认无参构造，但是会提供默认拷贝构造；如果用户定义拷贝构造函数，c++不会再提供其它的构造函数
- 析构函数：用来清理，函数名和类名相同，不能有参数，不能发生重载，对象在**销毁**前会调用析构函数，而且只会调用一次

- 1 | ~类名 () {}

- 构造函数和析构函数都是必须有的，若自己不定义系统会自动提供

```

1  class Person
2  {
3  public:
4      Person(const Person &p)//拷贝构造函数
5      {
6          cout << "person构造函数调用";
7      }
8      ~Person()
9      {
10         cout << "hhhhh" << endl;
11     }
12 };

```

## 深拷贝和浅拷贝

- 深拷贝和浅拷贝

- 浅拷贝：简单的赋值拷贝操作，浅拷贝带来的问题就是会使得堆区的内存重复释放；若利用系统提供的拷贝构造函数会使用浅拷贝
- 深拷贝：在堆区重新申请空间进行拷贝操作
- 初始化列表：c++提供初始化列表，对属性进行初始化

```

1  class Person
2  {
3  public:
4      Person(int a, int b, int c):m_a(a),m_b(b),m_c(c)
5      {
6
7      }
8      int m_a, m_b, m_c;
9  };

```

- 类对象作为类成员
- 静态成员 (static)
  - 静态成员变量
    - 所有对象共同分享一份数据
    - 在编译阶段分配内存（程序没有运行前分配内存，在全局区里面）
    - 类内声明，类外初始化，必须有初始值

```

1  class Person
2  {
3  public:
4      static int m_A;
5  };
6  int Person::m_A = 4;

```

- 可以通过对象进行访问，或者通过类名进行访问

```

1  cout << p.m_A << endl;
2  cout << Person::m_A << endl;//由于所有d

```

-



- 静态成员函数：静态成员函数可以访问静态成员变量，不可以访问非静态成员变量（由于所有对象共享一份数据）
- 成员变量和成员函数是分开存储的，非静态成员变量属于类的对象上，静态成员变量/函数和非静态成员函数不属于类的对象上

## this指针

- this指针：不需要定义，直接使用即可，是隐含在每一个非静态成员函数内的一种指针
  - this指针指向的是被调用的成员函数所属的对象
  - 作用1是解决名称冲突

```
1 class Person
2 {
3 public:
4     Person(int age)
5     {
6         this->age = age;
7     }
8     int age;
9 };
```

- 作用2是返回对象本身用\*this:

```
1 #include<iostream>
2 using namespace std;
3 class Person
4 {
5 public:
6     Person(int age)
7     {
8         this->age = age;
9     }
10    Person& PersonAddPerson(Person& p)//注意返回的是引用值
11    {
12        this->age += p.age;
13        return *this;
14    }
15    int age;
16 };
17 int main()
18 {
19     Person p1(10);
20     Person p2(10);
21     p2.PersonAddPerson(p1).PersonAddPerson(p1);//
22     cout << "年龄为" << p2.age << endl;
23 }
```

- this指针本质是指针常量，指针的指向是不可以修改的
- 关键字mutable：特殊变量，即使在常函数中，值加上关键字mutable也可以进行修改
- 常函数：常函数内不可以修改成员属性，成员属性声明时加关键字mutable后在常函数中依然可以修改

```

1  class Person
2  {
3  public:
4      void showPerson() const//this修饰的this的指向，指向的值是不可以修改的
5      {
6
7      }
8      int age;
9      int
10 };

```

- 常对象：在对象前加上const变为常对象，对象中的变量不可以进行修改，常对象只能调用常函数，不能调用普通成员函数，因为普通成员函数的属性可以进行修改

## 友元函数

- 友元函数：friend，可以访问类的私有成员
  - 全局函数做友元函数

```

1  #include<iostream>
2  using namespace std;
3  class Building
4  {
5      friend void goodgay(Building* building);//友元函数，写到类里面就可以
6  public:
7      Building()
8      {
9          m_sittingroom = "客厅";
10         m_restroom = "卧室";
11     }
12     string m_sittingroom;
13 private:
14     string m_restroom;
15 };
16
17 void goodgay(Building *building)
18 {
19     cout <<"来了" << building->m_sittingroom << endl;
20     cout << "来了"<< building->m_restroom << endl;
21 }
22
23 int main()
24 {
25     Building b;
26     goodgay(&b);
27 }

```

- 类做友元函数：(类外写成员函数)

```

1  #include<iostream>
2  using namespace std;
3  class Building;
4  class GoodGay
5  {

```

```

6 public:
7
8     GoodGay();
9     void visit();
10    Building* building;
11 };
12 class Building
13 {
14 public:
15     friend GoodGay; //类做友元函数
16     Building();
17     string m_sittingroom;
18 private:
19     string m_restroom;
20 };
21 Building::Building() //类外写成员函数
22 {
23     m_sittingroom = "客厅";
24     m_restroom = "卧室";
25 }
26 GoodGay::GoodGay() //类外写成员函数
27 {
28     building = new Building; //堆区新建
29 }
30 void GoodGay::visit() //类外写成员函数
31 {
32     cout << "好基友正在访问" << building->m_sittingroom << endl;
33     cout << "好基友正在访问" << building->m_restroom << endl;
34 }
35 void test01()
36 {
37     GoodGay g;
38     g.visit();
39 }
40 int main()
41 {
42     test01();
43 }

```

- 成员函数做友元

```

1 #include<iostream>
2 using namespace std;
3 class Building;
4 class GoodGay
5 {
6 public:
7     GoodGay();
8     void visit(); //让visit函数访问Building中的私有成员
9     void visit2(); //不可以访问Building中的私有成员
10    Building* building;
11 };
12 class Building
13 {
14     friend void GoodGay::visit(); //成员函数作为类的友元函数
15 public:
16     Building();

```

```

17     string m_sittingroom;
18 private:
19     string m_restroom;
20 };
21 //类外实现成员函数
22 Building::Building()
23 {
24     m_sittingroom = "客厅";
25     m_restroom = "卧室";
26 }
27 GoodGay::GoodGay()
28 {
29     building = new Building;
30 }
31 void GoodGay::visit()
32 {
33     cout << "visit函数正在访问" << building->m_sittingroom << endl;
34     cout << "visit函数正在访问" << building->m_restroom << endl;
35 }
36 }
37 void GoodGay::visit2()
38 {
39     cout << "visit2函数正在访问" << building->m_sittingroom << endl;
40 }
41 void test01()
42 {
43     GoodGay GG;
44     GG.visit();
45 }
46 }
47 int main()
48 {
49     test01();
50 }

```

## 运算符重载

- 运算符重载：给运算符重新进行定义，赋予其另外的功能，以适应不同的数据类型
  - 加号运算符重载：作用是实现两个自定义数据类型相加的运算;

```

1 #include<iostream>
2 using namespace std;
3 class Person
4 {
5 public:
6     Person operator+(Person& p1)
7     {
8         Person temp;
9         temp.m_A = this->m_A + p1.m_A;
10        temp.m_B = this->m_A + p1.m_B;
11        return temp;
12    }
13    int m_A;
14    int m_B;
15 };
16
17 void test()

```

```

18 {
19     Person p1;
20     p1.m_A = 10;
21     p1.m_B = 10;
22     Person p2;
23     p2.m_A = 33;
24     p2.m_B = 44;
25     Person p3 = p1 + p2;
26     cout << "m_a" << p3.m_A << "m_b" << p3.m_B;
27 }
28 int main()
29 {
30     test();
31     system("pause");
32 }

```

- 全局函数重载+号
- 左移运算符重载：通常**不会使用成员函数**对左移运算符进行重载;左移运算符重载实现自定义数据类型的输出（一般配合private使用）

```

1 #include<iostream>
2 using namespace std;
3 class Person
4 {
5 public:
6
7     int m_A;
8     int m_B;
9 };
10
11 ostream & operator<<(ostream &cout,Person& p)//本质上是
    operator<<(cout,p),简化为cout<<p
12 {
13     cout << "m_A=" << p.m_A << "    m_B=" << p.m_B << endl;
14     return cout;//ostream为输出流，返回该数据类型后调用后可以使用endl进行换
    行
15 }
16 void test01()
17 {
18     Person p;
19     p.m_A = 10;
20     p.m_B = 20;
21     cout << p << endl;
22 }
23 int main()
24 {
25     test01();
26     system("pause");
27 }

```

- 递增数据运算符的重载:区分前置和后置递增运算符：加入一个int占位则系统认为是后置递增，没有这是前置递增运算符进行重载；后置递增**一定返回的是值**，前置递增返回的是引用&

```

1 #include<iostream>
2 using namespace std;
3 class Person
4 {

```

```

5     friend ostream& operator<<(ostream& cout, Person p);
6 public:
7     Person()
8     {
9         m_num = 0;
10    }
11    //重载前置++运算符
12    Person& operator++()//返回引用是为了对一个数据进行操作
13    {
14        m_num++;//先进行++运算
15        return *this;//再将自身进行返回
16    }
17    //重载后置++运算符
18    Person operator++(int)//加入一个int占位，系统认为是后置递增，后置递增一
    定返回的是值
19    {
20        Person temp = *this;
21        m_num++;
22        return temp;
23    }
24 private:
25     int m_num = 0;
26 };
27 //重载左移运算符
28 ostream& operator<<(ostream& cout, Person p1)
29 {
30     cout << p1.m_num;
31     return cout;
32 }
33 void test01()
34 {
35     Person p1;
36     cout << "前置递增运算符" << ++(++p1) << endl;
37     cout << p1 << endl;
38 }
39 void test02()
40 {
41     Person pp1;
42     cout << "后置递增运算符" << pp1++ << endl;
43     cout << "后置递增运算符" << pp1 << endl;
44 }
45 int main()
46 {
47     test01();
48     test02();
49     system("pause");
50 }

```

#### ○ 赋值运算符的重载

- c++编译器至少对一个类添加4个函数
  - 默认构造函数
  - 默认析构函数
  - 默认拷贝构造函数
  - 赋值运算符operator=, 对属性进行值拷贝（如果类中有属性指向堆区，做赋值操作时也会出现深浅拷贝问题）

```

1  #include<iostream>
2  using namespace std;
3  class Person
4  {
5  public:
6      Person(int age)
7      {
8          m_age = new int(age); //在堆区开辟内存
9      }
10     //赋值运算符的重载
11     Person& operator=(Person &p)
12     {
13         //编译器默认的赋值运算符是浅拷贝
14         //判断是否有属性在堆区，清理干净，然后再深拷贝
15         if (m_age != NULL)
16         {
17             delete m_age;
18             m_age = NULL;
19         }
20         m_age = new int(*p.m_age); //深拷贝
21         return *this; //返回对象本身为了实现连等操作
22     }
23     int* m_age;
24
25 private:
26
27 };
28 void test01()
29 {
30     Person p1(70);
31     Person p2(30);
32     p1 = p2;
33     cout << *p1.m_age << endl;
34 }
35 int main()
36 {
37     test01();
38     system("pause");
39 }

```

○ 关系运算符的重载（大于号小于号等于号）

```

1  #include<iostream>
2  using namespace std;
3  class Person
4  {
5  public:
6      Person(string name, int age)
7      {
8          m_age = age;
9          m_name = name;
10     }
11     //重载关系运算符==
12     bool operator==(Person& p)
13     {
14         if (this->m_name == p.m_name && this->m_age == p.m_age)
15             return true;

```

```

16         else
17         {
18             return false;
19         }
20     }
21
22 private:
23     string m_name;
24     int m_age;
25 };
26
27 int main()
28 {
29     Person p1("zhangsan", 34);
30     Person p2("zhangsan", 34);
31     if (p1 == p2)
32     {
33         cout << "p1和p2相等" << endl;
34     }
35     else
36     {
37         cout << "p1和p2不相等" << endl;
38     }
39 }

```

- 函数调用运算符进行重载（函数调用运算符（）也可以进行重载）；由于重载后的使用方式非常像函数的调用，因此称为仿函数；仿函数没有固定的写法，非常灵活

```

■ 1  #include<iostream>
2  using namespace std;
3  class MyPrint
4  {
5  public:
6      //重载函数调用运算符
7      void operator()(string test)
8      {
9          cout << test << endl;
10     }
11
12 private:
13
14 };
15 class MyAdd
16 {
17 public:
18     int operator()(int num1, int num2)
19     {
20         return num1 + num2;
21     }
22 };
23 int main()
24 {
25     MyPrint p1;
26     p1("好家伙");//由于使用起来非常像函数调用，因此称为仿函数
27     MyAdd add;
28     cout << add(3, 5) << endl;
29     cout << MyAdd()(34, 67) << endl;//匿名函数对象的使用

```



- 匿名函数对象：当前对象使用完之后立即被释放

## 继承(面向对象的三大特性之一)

- 下级别的成员除了拥有上一级的共性，还有自己的特性
- 语法 class 子类（也称为派生类）：继承方式 父类（也称为基类）
- 派生类中包含基类中的成员（共性）以及自己新增的成员（个性）
- 作用是减少重复的代码
- 公共继承

```

1  #include<iostream>
2  using namespace std;
3  class BasePage
4  {
5  public:
6      void up()
7      {
8          cout << "gggg" << endl;
9      }
10     void down()
11     {
12         cout << "好家伙" << endl;
13     }
14
15 };
16 class JAVA: public BasePage
17 {
18 public:
19     void content()
20     {
21         cout << "lalal" << endl;
22     }
23 };
24 int main()
25 {
26     JAVA ja;
27     ja.down();
28     ja.content();
29     ja.up();
30 }

```

- 继承方式有三种：
  - 公共继承：在父类中的成员权限继承到子类中没有变化，父类的私有成员子类不可以访问
  - 私有继承：父类中的公共成员以及保护成员变成了子类中的私有成员，父类的私有成员不可以访问
  - 保护继承：父类中的公共成员以及保护成员在子类中变成了保护成员，私有成员子类不可以访问
- 父类中所有的非静态成员属性都会被子类继承下去，但是父类中私有成员属性是被编译器隐藏了，是访问不到，但是是被继承下去了
- 子类继承父类后，当创建子类对象，也会调用父类的构造函数

```

1  #include<iostream>
2  using namespace std;
3  class Base
4  {
5  public:
6      Base()
7      {
8          cout << "Base的构造函数" << endl;
9      }
10     ~Base()
11     {
12         cout << "Base的析构函数" << endl;
13     }
14
15
16 };
17 class Son : public Base
18 {
19 public:
20     Son()
21     {
22         cout << "son的构造函数" << endl;
23     }
24     ~Son()
25     {
26         cout << "son的析构函数" << endl;
27     }
28
29 private:
30
31 };
32 void test01()
33 {
34     Base b1;
35     Son s1;
36 }
37 int main()
38 {
39     test01();
40     system("pause");
41 }
42

```

- 先调用父类的构造函数，再调用子类的构造函数，然后调用子类的析构函数，再调用父类的析构函数
- 继承中同名成员处理方式
  - 访问子类同名成员，直接访问即可
  - 访问父类同名成员，需要加上作用域

```

1  #include<iostream>
2  using namespace std;
3  class Base
4  {
5  public:
6      Base()

```

```

7      {
8          cout << "Base的构造函数" << endl;
9      }
10     int m_a = 100;
11     ~Base()
12     {
13         cout << "Base的析构函数" << endl;
14     }
15
16
17 };
18 class Son : public Base
19 {
20 public:
21     Son()
22     {
23         cout << "son的构造函数" << endl;
24     }
25     int m_a = 34;
26     ~Son()
27     {
28         cout << "son的析构函数" << endl;
29     }
30
31 private:
32
33 };
34 void test01()
35 {
36     Son s1;
37     cout << "son下的m_a" << s1.m_a << endl;
38     cout << "Base下的m_a" << s1.Base::m_a << endl; //加上作用域即可访问
39 }
40 int main()
41 {
42     test01();
43     system("pause");
44 }
45

```

- 继承中同名成员函数的处理（同样是加作用域）

```

○ 1  #include<iostream>
2  using namespace std;
3  class Base
4  {
5  public:
6      void func()
7      {
8          cout << "base func" << endl;
9      }
10
11
12 };
13 class Son : public Base
14 {
15 public:

```

```

16     void func()
17     {
18         cout << "son func" << endl;
19     }
20     int m_a = 34;
21
22
23 private:
24
25 };
26 void test01()
27 {
28     Son s1;
29     s1.func();
30     s1.Base::func(); //加作用域
31 }
32 int main()
33 {
34     test01();
35     system("pause");
36 }
37

```

```

o 1  #include<iostream>
2  using namespace std;
3
4  class base
5  {
6  public:
7      base(int a, int b) { x = a; y = b; cout << "base constructor"
      << endl; }
8  private:
9      int x;
10     int y;
11 };
12
13 class derived : public base
14 {
15 public:
16     derived(int a, int b, int c) :base(a, b) { z = c; cout <<
      "derived constructor" << endl; }
17 private:
18     int z;
19 };
20
21 int main()
22 {
23     derived B(1, 2, 3);
24     return 0;
25 }

```

- 如果子类中出现和父类的同名成员函数，子类同名成员会隐藏掉父类中所有的同名成员函数，包括重载的情况，如果想访问一定要加作用域
- 总结：
  - 子类对象可以直接访问到子类中的同名成员

- 子类对象加作用域可以访问到父类的同名成员
- 当子类中含有与父类相同名称的成员函数，子类会隐藏父类中的同名成员函数，加上作用域可以访问到父类中的同名成员函数。
- 继承中的同名静态成员处理方式

```

○ 1  #include<iostream>
2  #include<time.h>
3  #include <stdlib.h>
4  using namespace std;
5  class Base
6  {
7  public:
8      static int m_a;
9      static void func()
10     {
11         cout << "Base中的func" << endl;
12     }
13 };
14 int Base::m_a = 10;
15
16 class Son:public Base
17 {
18 public:
19     static int m_a;
20     static void func()
21     {
22         cout << "Son中的func" << endl;
23     }
24 };
25 int Son::m_a = 1000;
26 void test01()
27 {
28     //1.通过对象访问
29     Son s;
30     cout << "通过对象访问" << endl;
31     cout << "son下m_a" << s.m_a << endl;
32     cout << "Base下m_a" << s.Base::m_a << endl;
33     //2.通过类名进行访问
34     cout << "通过类名进行访问" << endl;
35     cout << "son下m_a" << Son::m_a << endl; //通过类名进行访问
36     cout << "Base下m_a" << Son::Base::m_a << endl; //第一个::代表通过类
名进行访问，第二个::代表访问父类作用域下
37 }
38 void test02()
39 {
40     //1.通过对象对静态成员函数进行访问
41     Son s1;
42
43     s1.func();
44     cout << "通过对象访问" << endl;
45     s1.Base::func();
46     //2.通过类名对静态成员函数进行访问
47     cout << "通过类名进行访问" << endl;
48     Son::func();
49     Son::Base::func();
50 }
51 int main()

```

```

52 {
53     test01();
54     test02();
55     system("pause");
56 }

```

- 总结：同名静态成员处理方式和非静态处理方式一样，只不过有两种访问的方式（通过对象和通过类名）
- 多继承语法：c++允许一个类继承多个类
  - 语法：class 子类：继承方式 父类，继承方式 父类
  - 当父类中出现同名成员，需要加作用域区分
  - 通常在开发中不使用多继承语法
- 菱形继承：有两个派生类继承同一个基类，又有某个类同时继承两个派生类
  - 菱形继承导致最后那个类含有两份基类数据，导致数据空间浪费
  - 解决方式：继承之前加上关键字virtual变为虚继承
  - vbptr——虚基类指针，指向虚基类表，虚继承继承了指针

```

1  #include<iostream>
2  #include<time.h>
3  #include <stdlib.h>
4  using namespace std;
5  class animal
6  {
7  public:
8      int m_Age = 100;
9
10 };
11 class sheep : virtual public animal//animal为虚基类
12 {
13 public:
14     int m_name;
15 };
16 class Tuo :virtual public animal
17 {
18 public:
19     int tuo_name;
20 };
21
22 class Sheeptuo : public sheep, public Tuo
23 {
24 public:
25     int name21;
26 };
27 int main()
28 {
29     Sheeptuo p1;
30     cout << p1.m_Age<<endl; //直接访问不会出错
31     cout << p1.sheep::m_Age << endl;
32     system("pause");
33 }

```

# 多态 (c++面向对象三大特性之一)

- 多态分为两类：
  - 静态多态：函数重载和运算符重载属于静态多态，复用函数名
  - 动态多态：派生类和虚函数实现运行时的多态
- 静态多态和动态多态的区别：
  - 静态多态函数地址早绑定—编译器阶段就确定了函数地址
  - 动态多态的函数地址晚绑定—运行阶段确定函数地址
- 动态多态满足条件：
  - 有继承关系
  - 子类需要重写父类中的虚函数（**重写需要返回值类型，函数名称以及参数列表完全相同**）
- 动态多态的使用：父类的指针或者引用指向子类对象

```
1  #include<iostream>
2  #include<time.h>
3  #include <stdlib.h>
4  using namespace std;
5  class animal
6  {
7  public:
8      virtual void speak()//虚函数实现地址晚绑定
9      {
10         cout << "animal is speaking" << endl;
11     }
12 };
13 class Cat : public animal
14 {
15 public:
16     void speak()//!!
17     {
18         cout << "Cat is speaking" << endl;
19     }
20 };
21 void dospeak(animal& Animal)//由于地址早绑定，在编译阶段就确定了函数地址，所以是
    执行animal speak
22 {
23     Animal.speak();
24 }
25 void test01()
26 {
27     Cat cat;
28     dospeak(cat);//由于speak多态，传入什么对象就访问什么地址
29 }
30 //如果想执行cat speak,那么这个函数地址就不能提前绑定，需要使用多态进行晚绑定
31 int main()
32 {
33     test01();
34 }
```

- 多态的优点
  - 代码组织清晰
  - 可读性强
  - 利于前期和后期的扩展以及维护
- 在实际开发中提倡开闭原则：对扩展进行开放，对修改进行关闭

o

```
1  #include<iostream>
2  #include<time.h>
3  #include <stdlib.h>
4  #define MAXSIZE 100
5  using namespace std;
6  //实现计算器抽象类//任何功能都没有
7  class AbstractCalculator
8  {
9  public:
10     virtual int getResult()
11     {
12         return 0;
13     }
14     int m_num1;
15     int m_num2;
16 };
17 class AddCalculator : public AbstractCalculator
18 {
19 public:
20     int getResult()
21     {
22         return m_num1 + m_num2;
23     }
24 private:
25 };
26 class MulCalculator :public AbstractCalculator
27 {
28 public:
29     int getResult()
30     {
31         return m_num1 * m_num2;
32     }
33 private:
34 };
35 };
36 void test()
37 {
38     //多态使用条件
39     //父类指针或者引用指向子类对象
40     //加法运算
41     AbstractCalculator* p1 = new AddCalculator;
42     p1->m_num1 = 12;
43     p1->m_num2 = 23;
44     cout << p1->m_num1 << "+" << p1->m_num2 << "=" << p1-
45 >getResult() << endl;
46     delete p1; //堆区数据记得释放
47     AbstractCalculator* p2 = new MulCalculator;
48     p2->m_num1 = 12;
49     p2->m_num2 = 23;
50     cout << p2->m_num1 << "*" << p2->m_num2 << "=" << p2-
51 >getResult() << endl;
52 }
53 int main()
54 {
55     test();
56     system("pause");
57 }
```



- 纯虚函数和抽象类

- 在多态中，通常父类中的虚函数的实现是无意义的，主要是调用子类重写内容，因此可以将虚函数改为纯虚函数
- **纯虚类没有构造函数**
- 纯虚函数语法：virtual 返回值类型 函数名（参数列表） = 0；当类中有了纯虚函数，这个类也称为抽象类
- 抽象类的特点：
  - 无法实例化对象
  - 子类必须重写抽象类中的纯虚函数，否则也属于抽象类

```
1  #include<iostream>
2  using namespace std;
3  class Base
4  {
5  public:
6      virtual void func() = 0; //只要有一个纯虚函数，这个类就是抽象类,无法实例化
7  };
8  class Person : public Base
9  {
10 public:
11     void func()
12     {
13         cout << "好家伙" << endl;
14     }
15 };
16
17 void test()
18 {
19     Base* p1 = new Person;
20     p1->func();
21 }
22 int main()
23 {
24     test();
25     system("pause");
26 }
```

- 多态案例

```
1  #include<iostream>
2  #include<time.h>
3  #include <stdlib.h>
4  #define MAXSIZE 100
5  using namespace std;
6  //案例：制作饮品
7  class AbstractDrinking
8  {
9  public:
10     //煮水
11     virtual void Boil()= 0;
12     //冲泡
13     virtual void Brew() = 0;
14     //倒入杯中
15     virtual void PourInCup() = 0;
```

```
16     //加入辅料
17     virtual void PutSth() = 0;
18     //制作饮品
19     void makeDrink()
20     {
21         Boil();
22         Brew();
23         PourInCup();
24         PutSth();
25     }
26 private:
27
28 };
29 class Coffee : public AbstractDrinking
30 {
31 public:
32     virtual void Boil()
33     {
34         cout << "煮农夫山泉" << endl;
35     }
36     virtual void Brew()
37     {
38         cout << "冲泡咖啡" << endl;
39     }
40     virtual void PourInCup()
41     {
42         cout << "倒入杯中" << endl;
43     }
44     virtual void PutSth()
45     {
46         cout << "加糖" << endl;
47     }
48 private:
49
50 };
51 //制作茶叶
52 class Tea : public AbstractDrinking
53 {
54 public:
55     virtual void Boil()
56     {
57         cout << "煮矿泉水" << endl;
58     }
59     virtual void Brew()
60     {
61         cout << "冲茶叶" << endl;
62     }
63     virtual void PourInCup()
64     {
65         cout << "倒入杯中" << endl;
66     }
67     virtual void PutSth()
68     {
69         cout << "加入枸杞" << endl;
70     }
71 private:
72
73 };
```

```

74 void dowork(AbstractDrinking* abs)
75 {
76     abs->makeDrink();
77     delete abs;
78 }
79 void test01()
80 {
81     //制作咖啡
82     dowork(new Coffee);
83     //制作茶叶
84     dowork(new Tea);
85     //多态：一种接口有多种形态
86 }
87 int main()
88 {
89     test01();
90     system("pause");
91 }

```

## 虚析构和纯虚析构

- 多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码，解决方式是将父类中的析构函数改为虚析构或者纯虚析构
- 虚析构和纯虚析构的共性
  - 可以解决父类指针释放子类对象
  - 都需要有具体的函数实现
- 虚析构和纯虚析构区别
  - 如果是**纯虚析构**，该类属于抽象类，无法实例化对象
- 虚析构语法：virtual ~类名(){}
- 纯虚析构语法：virtual ~类名()=0; 类名:: ~类名()
- 纯虚析构函数**既要有函数的声明，也要有代码的实现**（得单独写出来）
- 如果子类中没有堆区数据，可以不写为虚析构或者纯虚析构

- ```

1  #include<iostream>
2  using namespace std;
3  //虚析构和纯虚析构
4  class Animal
5  {
6  public:
7      Animal()
8      {
9          cout << "Animal构造函数调用" << endl;
10     }
11     /*virtual ~Animal()
12     {
13         cout << "Animal析构函数调用" << endl;
14     }*/
15
16     virtual void speak() = 0;
17
18 private:
19
20 };
21 Animal:: ~Animal()
            
```

```

22 {
23     cout << "Animal纯虚析构函数调用"<<endl;
24 }
25 class Cat : public Animal
26 {
27 public:
28     Cat(string name)
29     {
30         m_name = new string(name); //堆区数据应该在析构函数释放掉
31     }
32     virtual void speak()
33     {
34         cout << *m_name << "小猫在说话" << endl;
35     }
36     string* m_name;
37     ~Cat()
38     {
39         if (m_name != NULL)
40         {
41             cout << "Cat析构函数调用" << endl;
42             delete m_name;
43             m_name = NULL;
44         }
45     }
46 };
47 void test01()
48 {
49     Animal* animal = new Cat("Tom");
50     animal->speak();
51     delete animal;
52 }
53 int main()
54 {
55     test01();
56     system("pause");
57 }

```

- 多态案例：电脑组装

```

1  #include<iostream>
2  using namespace std;
3  class CPU
4  {
5  public:
6      virtual void calculate() = 0;
7  private:
8
9  };
10 class GPU
11 {
12 public:
13     virtual void display() = 0;
14 private:
15
16 };
17 class RAM
18 {

```

```

19 public:
20     virtual void storage()= 0;
21
22 private:
23
24 };
25 class computer
26 {
27 public:
28     computer(CPU* cpu, GPU* gpu, RAM* ram)
29     {
30         m_cpu = cpu;
31         m_gpu = gpu;
32         m_ram = ram;
33     }
34     void work()
35     {
36         m_cpu->calculate();
37         m_gpu->display();
38         m_ram->storage();
39     }
40     //提供析构函数释放
41     ~computer()
42     {
43         //释放cpu零件
44         if (m_cpu != NULL)
45         {
46             delete m_cpu;
47             m_cpu = NULL;
48         }
49         //释放gpu零件
50         if (m_gpu != NULL)
51         {
52             delete m_gpu;
53             m_gpu = NULL;
54         }
55         //释放ram零件
56         if (m_ram != NULL)
57         {
58             delete m_ram;
59             m_ram = NULL;
60         }
61     }
62 private:
63     CPU* m_cpu;
64     GPU* m_gpu;
65     RAM* m_ram;
66 };
67 //具体厂商
68 class IntelCPU : public CPU
69 {
70 public:
71     virtual void calculate()
72     {
73         cout << "IntelCpu is working" << endl;
74     }
75
76 private:

```

```
77
78 };
79 class IntelGPU :public GPU
80 {
81 public:
82     virtual void display()
83     {
84         cout << "IntelGpu is working" << endl;
85     }
86
87 private:
88
89 };
90 class IntelRAM :public RAM
91 {
92 public:
93     virtual void storage()
94     {
95         cout << "IntelRAM is working" << endl;
96     }
97
98 private:
99
100 };
101 //lenovo
102 class LenovoCPU :public CPU
103 {
104 public:
105     virtual void calculate()
106     {
107         cout << "LenovoCpu is working" << endl;
108     }
109
110 private:
111
112 };
113 class LenovoGPU :public GPU
114 {
115 public:
116     virtual void display()
117     {
118         cout << "LenovoGpu is working" << endl;
119     }
120
121 private:
122
123 };
124 class LenovoRAM :public RAM
125 {
126 public:
127     virtual void storage()
128     {
129         cout << "LenovoRAM is working" << endl;
130     }
131
132 private:
133
134 };
```

```

135 void test01()
136 {
137     //第一台电脑零件
138     CPU* intelcpu = new IntelCPU;
139     GPU* intelgpu = new IntelGPU;
140     RAM* intelram = new IntelRAM;
141     computer* computer1 = new computer(intelcpu, intelgpu, intelram);
142     computer1->work();
143     delete computer1;
144     cout << "-----" << endl;
145     computer* computer2 = new computer(new LenovoCPU, new LenovoGPU,
new LenovoRAM);
146     computer2->work();
147     delete computer2;
148 }
149 int main()
150 {
151     test01();
152     system("pause");
153 }

```

## 文件操作

- 程序运行时产生的数据都属于临时数据，程序一旦运行结束都会被释放，通过文件可以将数据持久化
- c++中对文件操作需要包含头文件
- 文件类型分为两种
  - 文本文件：文件以文本的ascii码形式存储在计算机中
  - 二进制文件：文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂他们
- 操作文件分为三大类：
  - ofstream:写操作 (output file)
  - ifstream:读操作(input file)
  - fstream:读写操作
- **写文本文件的步骤\*\***
  1. 包含头文件 #include
  2. 创建流对象 ofstream ofs
  3. 打开文件 ofs.open("文件路径", 打开方式);
  4. 写数据 ofs<<"写入数据"
  5. 关闭文件 ofs.close()
- 文件打开方式：

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置在文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在先删除，再创建
ios::binary	二进制方式

- 文件打开方式可以配合使用，利用|操作符；如 利用二进制方式写文件：ios::binary | ios::out

- ```
1  #include<iostream>
2  #include<fstream>//包含头文件
3  using namespace std;
4  void test01()
5  {
6      //创建流对象
7      ofstream ofs;
8      //指定打开方式
9      ofs.open("text.txt", ios::binary|ios::out);
10     //写内容
11     ofs << "姓名: 张三" << endl;
12     ofs.close();
13 }
14
15 int main()
16 {
17     test01();
18 }
```

- **读文件步骤：**

1. 包含头文件 #include
2. 创建流对象 ifstream ifs
3. 打开文件并判断文件是否打开成功 ifs.open("文件路径", 打开方式);
4. 读数据：四种方式读数据
5. 关闭文件: ifs.close();

- ```
1  #include<iostream>
2  #include<fstream>//包含头文件
3  #include<string>
4  using namespace std;
5  void test01()
6  {
7      //创建流对象
8      ifstream ifs;
9      //打开文件并判断是否打开成功
10     ifs.open("text.txt",ios::in);
11     if (!ifs.is_open())
12     {
13         cout << "文件打开失败" << endl;
14     }
15     //读数据
16     ////第一种
17     //char buf[1024] = { 0 };
18     //while (ifs >> buf)
19     //{
20     //    cout << buf << endl;
21     //}
22     //ifs.close();
23     ////第二种
24     //char buf[1024] = {0};
25     //while (ifs.getline(buf,sizeof(buf)))
26     //{
27     //    cout << buf << endl;
28     //}
```



```

29     //第三种
30     /*string buf;
31     while (getline(ifs, buf))
32     {
33         cout << buf << endl;
34     }*/
35     //第四种(不建议使用)
36     char c;
37     while ((c = ifs.get()) != EOF)//EOF end of file
38     {
39         cout << c;
40     }
41 }
42 int main()
43 {
44     test01();
45 }

```

- 二进制文件：以二进制的方式对文件进行读写操作，打开方式指定为ios::binary

```

o 1  #include<iostream>
2  #include<fstream>//包含头文件
3  #include<string>
4  using namespace std;
5  class Person
6  {
7  public:
8      char m_name[64];//姓名、
9      int age;
10
11  private:
12
13  };
14  void test01()
15  {
16      //创建流对象
17      ofstream ofs("person.txt", ios::out | ios::binary);
18      Person p = { "张三", 18 };
19      //写文件
20      ofs.write((const char*)&p, sizeof(Person));
21      //关闭文件
22      ofs.close();
23  }
24  int main()
25  {
26      test01();
27  }

```

```

o 1  #include<iostream>
2  #include<fstream>//包含头文件
3  #include<string>
4  using namespace std;
5  //二进制文件的读取
6  class Person
7  {
8  public:
9      char m_name[64];//姓名、

```

```

10     int age;
11
12     private:
13
14 };
15 void test01()
16 {
17     //创建流对象
18     ifstream ifs;
19     Person p = { "张三", 18 };
20     //打开文件, 判断文件是否打开成功
21     ifs.open("Person.txt", ios::in | ios::binary);
22     if (!ifs.is_open())
23     {
24         cout << "文件打开失败" << endl;
25         return;
26     }
27     //读文件
28     Person p1;
29     ifs.read((char*)&p1, sizeof(Person));
30     //关闭文件
31     cout << "姓名:" << p1.m_name << "年龄: " << p1.age << endl;
32     ifs.close();
33 }
34 int main()
35 {
36     test01();
37 }

```

o

- 二级指针

## 函数模板

- C++提供另外一种编程思想称为**泛型编程**，主要利用的技术就是模板
- C++有函数模板和类模板
- 函数模板作用：建立一个通用函数，其函数返回值类型和形参类型可以不具体制定，用一个**虚拟的类型**来代表
- 语法

```
1 template<typename T>
```

template-声明创建模板

typename-表明其后面的符号是一种数据类型，可以用class代替

T-通用数据类型，名称可以替换，通常为大写字母

- ```

1 #include<iostream>
2 using namespace std;
3 //函数模板
4 template<typename T> //声明一个模板, 告诉编译器后面代码中紧跟着的T不要报错, T代表
  一个通用数据类型
5 void swap1(T& a, T& b)
6 {
7     T temp;

```

```

8     temp = a;
9     a = b;
10    b = temp;
11 }
12 template<typename T>
13 void swap2(T& a, T& b)
14 {
15     T temp;
16     temp = a;
17     a = b;
18     b = temp;
19 }
20 void test01()
21 {
22     int a = 10;
23     int b = 20;
24     swap1(a, b); //自动类型推导
25     swap2<int>(a, b); //显示指定类型
26     cout << "a=" << a
27           << "b=" << b << endl;
28 }
29 int main()
30 {
31     test01();
32     system("pause");
33     return 0;
34 }

```

- 总结：模板的作用是类型参数化，提高了代码的复用性；函数模板利用关键字template；函数模板有两种使用方式：自动类型推导以及显示指定类型
- 注意事项：
  - 自动类型推导必须推导出一致的数据类型T才可以使用
  - 模板必须要确定出T的数据类型才可以使用

```

1     template<class T>
2     void func()
3     {
4         cout << "hhhh" << endl;
5     }
6     int main()
7     {
8         func<int>();
9         system("pause");
10        return 0;
11    }

```

```

1     #include<iostream>
2     using namespace std;
3     //函数模板
4     //实现通用 对数组进行排序的函数
5     template<class T>
6     void mySwap(T& a, T& b)
7     {
8         T temp = a;
9         a = b;

```

```

10     b = temp;
11 }
12 template <class T>
13 void printArray(T arr[], int len)
14 {
15     for (int i = 0; i < len; i++)
16     {
17         cout << arr[i]<<" ";
18     }
19     cout << endl;
20 }
21 template <class T>
22 void mySort(T arr[], int len)
23 {
24     for (int i = 0; i < len; i++)
25     {
26         int max = i;
27         for (int j = i + 1; j < len; j++)
28         {
29             if (arr[max] < arr[j])
30             {
31                 max = j;
32             }
33         }
34         if (max != i)
35         {
36             mySwap(arr[max], arr[i]);
37         }
38     }
39 }
40 void test01()
41 {
42     char charr[] = "gfreewdg";
43     int num = sizeof(charr) / sizeof(char);
44     mySort(charr, num);
45     printArray(charr, num);
46 }
47 void test02()
48 {
49     int arrint[] = {2,4,6,1,7,3};
50     int num = sizeof(arrint) / sizeof(int);
51     mySort(arrint, num);
52     printArray(arrint,num);
53 }
54
55 int main()
56 {
57     test01();
58     test02();
59     system("pause");
60 }

```

- 普通函数与函数模板区别

- 普通函数可以发生自动类型转换（隐式类型转换）
- 函数模板调用时，利用自动类型推导，不会发生隐式类型转换
- 如果利用显示指定类型方式，可以发生隐式类型转换

- ```

1  #include<iostream>
2  using namespace std;
3  //函数模板
4  template <class T>
5  T myAdd01(T a, T b)
6  {
7      return a + b;
8  }
9  void test01()
10 {
11     int a = 10;
12     char b = 's';
13     cout << myAdd01<int>(a, b);
14 }
15 int main()
16 {
17     test01();
18 }

```

- 普通函数与函数模板的调用规则

- 如果函数模板和普通函数都可以实现，**优先调用普通函数**

- ```

1  #include<iostream>
2  using namespace std;
3  //函数模板与普通函数的调用规则
4  void myPrint(int a, int b); //此时也是调用普通函数，报错
5  template<class T>
6  void myPrint(T a, T b)
7  {
8      cout << "调用模板函数" << endl;
9  }
10 void test01()
11 {
12     int a = 19;
13     int b = 29;
14     myPrint(a, b);
15 }
16 int main()
17 {
18     test01();
19 }

```

- 可以通过空模板参数列表来强制调用函数模板

- ```

1  #include<iostream>
2  using namespace std;
3  //函数模板与普通函数的调用规则
4  void myPrint(int a, int b);
5
6  template<class T>
7  void myPrint(T a, T b)
8  {
9      cout << "调用模板函数" << endl;
10 }
11 void test01()
12 {

```

```

13     int a = 19;
14     int b = 29;
15     myPrint<>(a, b); //强制调用函数模板
16 }
17 int main()
18 {
19     test01();
20 }

```

- 函数模板也可以发生重载

```

1  #include<iostream>
2  using namespace std;
3  //函数模板与普通函数的调用规则
4  void myPrint(int a, int b);
5  //{
6  //  cout << "调用普通函数" << endl;
7  //}
8  template<class T>
9  void myPrint(T a, T b)
10 {
11     cout << "调用模板函数" << endl;
12 }
13 template<class T>
14 void myPrint(T a, T b, T c)
15 {
16     cout << "调用重载的模板函数" << endl;
17 }
18 void test01()
19 {
20     int a = 19;
21     int b = 29;
22     int c = 32;
23     myPrint<>(a, b, c);
24 }
25 int main()
26 {
27     test01();
28 }

```

- 如果函数模板可以产生更好的匹配，优先调用函数模板

```

1  #include<iostream>
2  using namespace std;
3  //函数模板与普通函数的调用规则
4  void myPrint(int a, int b)
5  {
6     cout << "调用普通函数" << endl;
7 }
8  template<class T>
9  void myPrint(T a, T b)
10 {
11     cout << "调用模板函数" << endl;
12 }
13 void test01()
14 {
15     char a = 19;

```

```

16     char b = 29;
17     myPrint(a, b); //此时调用模板时不用转换数据类型，直接调用模板
18 }
19 int main()
20 {
21     test01();
22 }

```

- 在实际开发中，使用普通函数就不会使用函数模板
- 模板的局限性：模板的通用性并不是万能的，某些特定的数据类型需要用具体化方式做特殊实现

- ```

1  #include<iostream>
2  using namespace std;
3  //函数模板的局限性
4  class Person
5  {
6  public:
7      Person(string name, int age);
8      ~ Person();
9      string m_name;
10     int m_age;
11
12 private:
13
14 };
15
16 Person:: Person(string name, int age)
17 {
18     this->m_name = name;
19     this->m_age = age;
20 }
21
22 Person::~ ~ Person()
23 {
24 }
25 template<class T>
26 bool myCompare(T& a, T& b)
27 {
28     if (a == b)
29     {
30         return true;
31     }
32     else
33     {
34         return false;
35     }
36 }
37 //利用具体化的Person的版本实现代码，具体化优先调用
38 template<> bool myCompare(Person& a, Person& b)
39 {
40     if (a.m_name == b.m_name)
41     {
42         return true;
43     }
44     else
45     {
46         return false;

```

```

47     }
48 }
49 void test01()
50 {
51     int a = 10;
52     int b = 20;
53     bool ret = myCompare(a, b);
54     if (ret)
55     {
56         cout << "a == b" << endl;
57     }
58     else
59     {
60         cout << "a != b" << endl;
61     }
62     Person p1("liujun", 20);
63     Person p2("liujun", 23);
64     bool ret1 = myCompare(p1, p2);
65     cout << "具体化: " << ret1 << endl;
66 }
67 int main()
68 {
69     test01();
70     system("pause");
71 }

```

- 学习模板并不是为了写模板，而是为了在STL能够运用系统提供的模板

## 类模板

- 类模板的作用：建立一个通用类，类中的成员数据类型可以不具体指定，用一个虚拟的类型来代表
- 语法

- ```

1  template<typename T>
2  类

```

- 解释：

- template-声明创建模板
- typename-表明其后面的符号是一种数据类型，可以用class代替
- T--通用数据类型，名称可以替换，通常为大写字母

- ```

1  #include<iostream>
2  using namespace std;
3  //类模板
4  template <typename NameType, typename AgeType>
5  class Person
6  {
7  public:
8      Person(NameType name, AgeType age)
9      {
10         this->m_name = name;
11         this->m_age = age;
12     }
13     NameType m_name;
14     AgeType m_age;
15 };

```



```

16 void test01()
17 {
18     Person<string, int>p1("liujun", 34); //模板参数列表
19     cout << "姓名为" << p1.m_name << "\n"
20         << "年龄为" << p1.m_age << endl;
21 }
22 int main()
23 {
24     test01();
25 }

```

- 类模板与函数模板十分类似，在声明模板template后面加类，此类称为类模板
- 类模板与函数模板的区别

- 类模板没有自动类型推导的使用方式

- 类模板在模板参数列表中可以有默认参数

```

1 #include<iostream>
2 using namespace std;
3 //类模板
4 template <typename NameType, typename AgeType = int> //AgeType默认为整
   型
5 class Person
6 {
7 public:
8     Person(NameType name, AgeType age)
9     {
10         this->m_name = name;
11         this->m_age = age;
12     }
13     NameType m_name;
14     AgeType m_age;
15 };
16 void test01()
17 {
18     Person<string, int>p1("liujun", 34); //模板参数列表
19     cout << "姓名为" << p1.m_name << "\n"
20         << "年龄为" << p1.m_age << endl;
21 }
22 void test02()
23 {
24     Person<string>p2("孙悟空", 100); //有默认数据类型，可以不用声明
25 }
26 int main()
27 {
28     test01();
29     system("pause");
30 }

```

- 类模板中成员函数和普通类中成员函数创建时机是有区别的
  - 普通类中成员函数一开始就可以创建
  - 类模板中的成员函数在调用时才创建

```

1 #include<iostream>
2 using namespace std;
3 //类模板中成员函数的创建时机

```

```

4  class Person1
5  {
6  public:
7      void showPerson1()
8      {
9          cout << "Person1 show" << endl;
10     }
11 };
12 class Person2
13 {
14 public:
15     void showPerson2()
16     {
17         cout << "Person2 show" << endl;
18     }
19
20 private:
21
22 };
23 template<typename T>
24 class Show
25 {
26 public:
27     T obj;
28     //类模板中成员函数在调用时才创建
29     void func1()
30     {
31         obj.showPerson1();
32     }
33     void func2()
34     {
35         obj.showPerson2();
36     }
37
38 private:
39
40 };
41 void test01()
42 {
43     Show<Person1>m1;
44     m1.func1();
45 }
46 int main()
47 {
48     test01();
49 }

```

- 类模板对象做函数参数：类模板实例化出的对象；向函数传参的方式

- 向函数传参一共有三种方式

- 指定传入类型--直接显示对象的数据类型
- 参数模板化--将对象中的参数变为模板进行传递
- 整个类模板化--将这个对象类型模板化进行传递

- ```

1  #include<iostream>
2  using namespace std;
3  //类模板对象做函数参数

```

```

4  template<class T1, class T2>
5  class Person
6  {
7  public:
8      Person(T1 name, T2 age)
9      {
10         this->m_name = name;
11         this->m_age = age;
12     }
13     void showPerson()
14     {
15         cout << "姓名" << this->m_name
16             << "\n年龄" << this->m_age << endl;
17     }
18     T1 m_name;
19     T2 m_age;
20 private:
21
22 };
23 //指定传入类型,最常用的
24 void printPerson1(Person<string, int> &p)
25 {
26     p.showPerson();
27 }
28
29
30 void test01()
31 {
32     Person<string, int>p1("孙悟空", 100);
33     printPerson1(p1);
34 }
35 //参数模板化
36 template<class T1, class T2>
37 void printPerson2(Person<T1, T2>& p)
38 {
39     p.showPerson();
40     cout << "T1的数据类型为" << typeid(T1).name() << endl
41         << "T2的数据类型为" << typeid(T2).name() << endl;
42 }
43 void test02()
44 {
45     Person<string, int> p2("猪八戒", 43);
46     printPerson2(p2);
47 }
48 //将整个类模板化
49 template<class T>
50 void printPerson3(T& p)
51 {
52     p.showPerson();
53     cout << "T的数据类型为" << typeid(T).name() << endl;
54 }
55 void test03()
56 {
57     Person<string, int>P3("沙僧", 43);
58     printPerson3(P3);
59 }
60 int main()
61 {

```

```

62     test01();
63     test02();
64     test03();
65 }

```

- 类模板与继承

- 当子类继承的父类是一个类模板时，子类在声明的时候要指定出父类中T的类型
- 如果不指定，编译器无法给子类分配内存
- 如果想灵活指定出父类中T的类型，子类也需要变为类模板

```

1  #include<iostream>
2  using namespace std;
3  //类模板与继承
4  template<class T>
5  class Base
6  {
7  public:
8      T m;
9  };
10 class Son :public Base<int> //必须要知道父类中T的数据类型才能够继承给子类
11 {
12
13 };
14 void test01()
15 {
16     Son s1;
17 }
18 //如果想灵活指定父类中T类型，子类也需要变类模板
19 template<class T1, class T2>
20 class Son1 :public Base<T2>
21 {
22 public:
23     Son1()
24     {
25         cout << "T1的数据类型为" << typeid(T1).name()
26             << "\nT2的数据类型为" << typeid(T2).name() << endl;
27     }
28     T1 obj;
29 };
30 void test02()
31 {
32     Son1<int, char>s1;
33 }
34 int main()
35 {
36     test01();
37     test02();
38 }
39

```

- 类模板的成员函数的类外实现

```

1  #include<iostream>
2  using namespace std;
3  //类模板成员函数类外实现

```

```

4  template<class T1, class T2>
5  class Person
6  {
7  public:
8      Person(T1 name, T2 age);
9      void showPerson();
10     T1 m_name;
11     T2 m_age;
12 };
13 template<class T1, class T2>
14 Person<T1, T2>::Person(T1 name, T2 age)
15 {
16     this->m_name = name;
17     this->m_age = age;
18 }
19 template<class T1, class T2>
20 void Person<T1, T2>::showPerson()
21 {
22     cout << "姓名" << this->m_name
23         << "\t年龄" << this->m_age << endl;
24 }
25 void test01()
26 {
27     Person<string, int> P1("liujun", 43);
28     P1.showPerson();
29 }
30 int main()
31 {
32     test01();
33 }
34

```

- 类模板分文件编写

- 第一种解决方式：直接包含.cpp文件
- 第二种解决方式：将.h文件以及.cpp文件中的内容写到一起，将后缀名改为.hpp文件(实际开发中用的多)

- ```

1  // person.hpp
2  template<class T1, class T2>
3  Person<T1, T2>::Person(T1 name, T2 age)
4  {
5      this->m_name = name;
6      this->m_age = age;
7  }
8  template<class T1, class T2>
9  void Person<T1, T2>::showPerson()
10 {
11     cout << "姓名" << this->m_name
12         << "\t年龄" << this->m_age << endl;
13 }

```

```

1 //test1028.cpp
2 #include<iostream>
3 #include"person.hpp"
4 using namespace std;
5 void test01()
6 {
7     Person<string, int> P1("liujun", 43);
8     P1.showPerson();
9 }
10 int main()
11 {
12     test01();
13 }

```

- 类模板与友元函数

- 全局函数类内实现--直接在类内声明友元函数即可
- 全局函数类外实现-需要提前让编译器知道全局函数的存在

```

1 #include<iostream>
2 using namespace std;
3 template<class T1, class T2> //提前告诉编译器有person模板类
4 class Person;
5 //全局函数类外实现
6 template<class T1, class T2> //提前告诉编译器有全局函数实现
7 void printPerson2(Person<T1, T2> p)
8 {
9     cout << "姓名" << p.m_name
10         << "\t年龄" << p.m_age << endl;
11 }
12 template<class T1, class T2>
13 class Person
14 {
15     //全局函数在类内实现
16     friend void printPerson(Person<T1, T2> p)
17     {
18         cout << "姓名" << p.m_name
19             << "\t年龄" << p.m_age << endl;
20     }
21     friend void printPerson2<>(Person<T1, T2> p); //加空模板参数列表<>
    代表函数模板的声明
22 public:
23     Person(T1 name, T2 age)
24     {
25         this->m_name = name;
26         this->m_age = age;
27     }
28 private:
29     T1 m_name;
30     T2 m_age;
31 };
32
33
34 void test01()
35 {
36     Person<string, int> P1("liuj", 23);
37     printPerson(P1);

```

```

38
39 }
40
41 void test02()
42 {
43     Person<string, int>P2("hao", 32);
44     printPerson2(P2);
45 }
46 int main()
47 {
48     test01();
49     test02();
50     system("pause");
51 }

```

- 没有特殊情况都使用类内实现

- ```

1 myarray.hpp
2 #pragma once
3 #include<iostream>
4 using namespace std;
5 //数组类封装：类模板案例
6 template<class T>
7 class MyArray
8 {
9 public:
10     MyArray(int capacity)//有参构造
11     {
12         cout << "有参构造调用" << endl;
13         this->m_capacity = capacity;
14         this->m_size = 0;
15         this->pAddress = new T[this->m_capacity];
16     }
17     //拷贝构造
18     MyArray(const MyArray& arr)
19     {
20         cout << "拷贝构造调用" << endl;
21         this->m_capacity = arr.m_capacity;
22         this->m_size = arr.m_size;
23         //深拷贝
24         this->pAddress = new T[arr.m_capacity];
25         for (int i = 0; i < this->m_size; i++)
26         {
27             this->pAddress[i] = arr.pAddress[i];
28         }
29         //return *this;
30     }
31     //operator= 防止浅拷贝问题
32     MyArray& operator=(const MyArray& arr)
33     {
34         //先判断原来堆区是否有数据，如果有先释放
35         if (this->pAddress != NULL)
36         {
37             delete[] this->pAddress;
38             this->pAddress = NULL;
39             this->m_capacity = 0;
40             this->m_size = 0;

```

```

41     }
42     this->m_capacity = arr.m_capacity;
43     this->m_size = arr.m_size;
44     this->pAddress = new T[arr.m_capacity];
45     for (int j = 0; j < this->m_size; j++)
46     {
47         this->pAddress[j] = arr.pAddress[j];
48     }
49     return *this;
50 }
51 //尾插法
52 void Push_Back(const T &val)
53 {
54     //判断容量是否等于大小
55     if (this->m_capacity == this->m_size)
56     {
57         return;
58     }
59     this->pAddress[this->m_size] = val;
60     this->m_size++;
61 }
62 //尾删法
63 void Pop_Back()
64 {
65     //让用户访问不到最后一个元素
66     if (this->m_size == 0)
67     {
68         return;
69     }
70     this->m_size--;
71 }
72 //通过下标方式访问数组中的元素
73 T& operator[](int index)
74 {
75     return this->pAddress[index];
76 }
77 //返回数组容量
78 int getCapacity()
79 {
80     return this->m_capacity;
81 }
82 int getSize()
83 {
84     return this->m_size;
85 }
86 ~MyArray()
87 {
88     if (this->pAddress != NULL)
89     {
90         delete[] this->pAddress;
91         this->pAddress = NULL;
92     }
93     cout << "析构函数调用" << endl;
94 }
95
96 private:
97     T* pAddress; //指针指向堆区开辟真实数组
98     int m_capacity; //数组容量

```



```
99     int m_size;//数组大小
100
101 };
102 ///-----
103 ---//
104 //myarray.cpp
105 #include<iostream>
106 #include"myarray.hpp"
107 using namespace std;
108 void printIntArray(MyArray<int>& arr)
109 {
110     for (int i = 0; i < arr.getSize(); i++)
111     {
112         cout << arr[i] << endl;
113     }
114 }
115 void test01()
116 {
117     MyArray<int> arr1(3);//有参构造
118     for (int i = 0; i < 3; i++)
119     {
120         arr1.Push_Back(i);
121     }
122     printIntArray(arr1);
123     cout << "arr1的容量为" << arr1.getCapacity() << endl
124          << "arr1的大小为" << arr1.getSize() << endl;
125     MyArray<int> arr2(arr1);
126     cout << "arr2的打印输出为";
127     printIntArray(arr2);
128     arr2.Pop_Back();
129     printIntArray(arr2);
130     cout << "arr2的容量为" << arr2.getCapacity() << endl
131          << "arr2的大小为" << arr2.getSize() << endl;
132 }
133 //测试自定义数据类型
134 class Person
135 {
136 public:
137     Person() {}
138     Person(string name, int age)
139     {
140         this->m_name = name;
141         this->m_age = age;
142     }
143     ~Person() {}
144     string m_name;
145     int m_age;
146 private:
147
148 };
149 void printPersonArray(MyArray<Person>& arr)
150 {
151     for (int i = 0; i < arr.getSize(); i++)
152     {
153         cout << "姓名" << arr[i].m_name
154              << "\t年龄" << arr[i].m_age << endl;
155     }
```

```

156 }
157 void test02()
158 {
159     MyArray<Person>arr(10);
160     Person p1("赵云", 23);
161     Person p2("曹操", 32);
162     Person p3("诸葛亮", 54);
163     Person p4("司马懿", 30);
164     Person p5("吕布", 40);
165     //将数据插入到数组中
166     arr.Push_Back(p1);
167     arr.Push_Back(p2);
168     arr.Push_Back(p3);
169     arr.Push_Back(p4);
170     arr.Push_Back(p5);
171     printPersonArray(arr);
172     cout << "arr的容量" << arr.getCapacity() << endl;
173     cout << "arr的大小" << arr.getSize() << endl;
174 }
175 int main()
176 {
177     test01();
178     test02();
179 }
180

```

## STL

- 长久以来，软件界一直希望建立一种可以重复利用的东西，C++的**面向对象**和**泛型编程**思想，目的就是复用性提升；大多数情况下，数据结构和算法都未能有一套标准导致被迫从事大量重复性工作；为了建立数据结构和算法的一套标准，诞生了STL
- STL基本概念
  - STL (standard template library,标准模板库)
  - **STL从广义上分为容器，算法和迭代器**
  - STL几乎所有代码都采用了模板类或者模板函数
- STL**六大组件**
  - 容器：各种数据结构，如vector, list, deque, set, map等，用来存放数据
  - 算法：各种常用的算法，如sort, find, copy, for\_each等
  - 迭代器：扮演了容器与算法之间的胶合剂
  - 仿函数：行为类似函数，可作为算法的某种策略；重载()
  - 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
  - 空间适配器：负责空间的配置与管理
- 容器分为**序列式容器**和**关联式容器**
  - 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元素
  - 关联式容器：二叉树结构，各元素之间没有严格的物理上的顺序关系
- 算法：有限的步骤解决逻辑或数学上的问题
  - 质变算法：是指运算过程中会更改区间内的元素内容，如拷贝替换和删除
  - 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，遍历等
- 迭代器：提供一种方法，使之能够依序访问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
  - 每个容器都有自己专属的迭代器
  - 迭代器使用非常类似于指针，初学阶段可以将迭代器理解为指针

- 迭代器种类(常用的迭代器为**双向迭代器**和**随机访问迭代器**)

种类	功能	支持运算
输入迭代器	对数据的只读访问	只读, 支持++, ==, !=
输出迭代器	对数据的只写访问	只写, 支持++
前向迭代器	读写操作, 并能向前推进迭代器	读写, 支持++, ==, !=
双向迭代器	读写操作, 并能向前和向后操作	读写, 支持++, --
随机访问迭代器	读写操作, 可以以跳跃的方式访问任意数据, 功能最强的迭代器	读写, 支持++, ==, [n], -n, <, <=, >, >=

- ```

1  #include<iostream>
2  #include<vector>
3  #include<algorithm> //标准算法的头文件
4  using namespace std;
5  //vector容器存放内置数据类型
6  void myprint(int val)
7  {
8      cout << val << endl;
9  }
10 void test01()
11 {
12     //创建一个vector容器 (数组)
13     vector<int> v;
14     //向容器中插入数据
15     v.push_back(2);
16     v.push_back(6);
17     v.push_back(5);
18     v.push_back(9);
19     v.push_back(3);
20     ///通过迭代器访问容器中的数据
21     //vector<int>::iterator itBegin = v.begin(); //起始迭代器, 指向容器
    中第一个元素
22     //vector<int>::iterator itEnd = v.end(); //结束迭代器, 指向容器中最后
    一个元素的下一个位置
23     ///第一种遍历方式
24     //while (itBegin != itEnd)
25     //{
26     //    cout << *itBegin << endl;
27     //    itBegin++;
28     //}
29     /*****
30     //第二种遍历方式
31     /*for (vector<int>::iterator it = v.begin(); it != v.end();
    it++)
32     {
33         cout << *it << endl;
34     }*/
35     /*第三种遍历方式, 利用STL提供遍历算法*/

```

```

36     for_each(v.begin(), v.end(), myprint);
37 }
38 int main()
39 {
40     test01();
41 }

```

- vector容器嵌套容器

```

1  #include<iostream>
2  #include<vector>
3  #include<algorithm> //标准算法的头文件
4  using namespace std;
5  void test01()
6  {
7      vector<vector<int>> v;
8      vector<int> v1;
9      vector<int> v2;
10     vector<int> v3;
11     vector<int> v4;
12     for (int i = 0; i < 4; i++)
13     {
14         v1.push_back(i + 1);
15         v2.push_back(i + 2);
16         v3.push_back(i + 3);
17         v4.push_back(i + 4);
18     }
19     v.push_back(v1);
20     v.push_back(v2);
21     v.push_back(v3);
22     v.push_back(v4);
23     for (vector<vector<int>>::iterator it = v.begin(); it != v.end();
24         it++)
25     {
26         for (vector<int>::iterator it1 = (*it).begin(); it1 !=
27             (*it).end(); it1++)
28         {
29             cout << *it1 << "\t"; //容器嵌套
30         }
31         cout << endl;
32     }
33 }
34 int main()
35 {
36     test01();
37 }

```

- string基本概念

- 本质：string是c++的字符串，而string本质上是一个类
- string和char\*的区别
  - char\*是一个指针
  - string是一个类，类的内部封装了char,管理这个字符串，是一个char型的容器

- 特点

- string内部封装了很多成员方法，例如：查找find，拷贝copy，删除delete，替换replace，插入insert
- string管理char\*所分配的内存，不用担心复制越界和取值越界等，由类内部进行负责
- string构造函数

```

1  #include<iostream>
2  using namespace std;
3  void test01()
4  {
5      string s1;//默认构造，创建一个空字符串
6      const char* str = "hello world";
7      string s2(str); //使用字符串str初始化
8      cout << "s2=" << s2 << endl;
9      string s3(s2);
10     cout << "s3=" << s3 << endl;//使用一个string对象初始化另一个string对象
11     string s4(10, 'a');//使用N个字符c初始化
12     cout << "s4=" << s4 << endl;
13 }
14 int main()
15 {
16     test01();
17 }

```

- string赋值操作

```

1  #include<iostream>
2  using namespace std;
3  //string的赋值操作
4  void test01()
5  {
6      string str1;
7      str1 = "Hello world";//char*字符串赋值给当前字符串
8      cout << "str1=" << str1 << endl;
9      string str2;
10     str2 = str1;//把字符串s赋值给当前的字符串
11     cout << "str2= " << str2 << endl;
12     string str3;
13     str3 = 'a';//字符赋值给当前的字符串
14     cout << "str3=" << str3 << endl;
15     string str4;
16     str4.assign("Hello c++");//把字符串S赋值给当前的字符串
17     cout << "str4=" << str4 << endl;
18     string str5;
19     str5.assign("Hello c++", 5);//把字符串前n个字符赋值给字符串
20     cout << "str5=" << str5 << endl;
21     string str6;
22     str6.assign(str5); //把字符串s赋值给当前的字符串
23     cout << "str6=" << str6 << endl;
24     string str7;
25     str7.assign(10, 'w');//用N个字符赋值给字符串
26     cout << "str7=" << str7 << endl;
27 }
28 int main()
29 {
30     test01();
31 }

```

- string字符串的拼接

- 实现字符串末尾拼接字符串

- ```
1 #include<iostream>
2 using namespace std;
3 //string的拼接操作
4 void test01()
5 {
6     string str1 = "hi";
7     str1 += "world";//重载+=操作符
8     cout << "str1= " << str1 << endl;
9     str1 += 'a';//重载+=操作符
10    cout << "str1= " << str1 << endl;
11    string str2 = "internet";
12    str1 += str2;
13    cout << "str1= " << str1 << endl;
14    string str3 = "work";
15    str3.append("boring");
16    cout << "str3= " << str3 << endl;
17    str3.append("gameabcde", 4);
18    cout << "str3= " << str3 << endl;
19    str3.append(str2);
20    cout << "str3= " << str3 << endl;
21    str3.append(str2, 4, 3);//从0开始截取，截取三位字符
22    cout << "str3= " << str3 << endl;
23 }
24 int main()
25 {
26     test01();
27 }
```

- string查找和替换

- 查找：查找指定字符串是否存在

- 替换：在指定位置替换字符串

- ```
1 #include<iostream>
2 using namespace std;
3 //string的查找和替换
4 //1 查找
5 void test01()
6 {
7
8     string str1 = "abcdefg";
9     int pos = str1.find("de");//从0开始查找字符串中是否有de;若没有返回-1, 若有返回起始位置值
10    if (pos != -1)
11    {
12        cout << "字符串找到" << endl;
13    }
14    else
15    {
16        cout << "字符串不存在" << endl;
17    }
18    pos = str1.rfind("de");//rfind是从右往左查找，find是从左往右查找
19    cout << "pos" << pos << endl;
20 }
```

```

21 //2 替换
22 void test02()
23 {
24     string str2 = "abcdefg";
25     str2.replace(1, 3, "1111");//从第一个位置到第三个位置替换为字符串1111
26     cout << str2 << endl;
27 }
28 int main()
29 {
30     test01();
31     test02();
32 }

```

- string字符串比较

- 字符串的比较是按照字符的ascii码进行对比

- = 返回0 > 返回1 < 返回-1
    - 最大的用途是判断两个字符串之间是否相等

- ```

1 #include<iostream>
2 using namespace std;
3 //string的比较
4 void test01()
5 {
6     string str1 = "hello";
7     string str2 = "hello";
8     if (str1.compare(str2) == 0)
9     {
10         cout << "str1 = str2" << endl;
11     }
12     else if (str1.compare(str2) > 0)
13     {
14         cout << "str1>str2" << endl;
15     }
16     else if (str1.compare(str2) < 0)
17     {
18         cout << "str1 < str2" << endl;
19     }
20 }
21 int main()
22 {
23     test01();
24 }

```

- string字符存取（对单个字符进行读或者写）

- ```

1 #include<iostream>
2 using namespace std;
3 //string字符存取
4 void test01()
5 {
6     string str1 = "hello";
7     //1 通过[]访问单个字符
8     for (int i = 0; i < str1.size(); i++)
9     {
10         cout << str1[i] << " ";
11     }

```

```

12     cout << endl;
13     //2 通过at方式访问单个字符
14     for (int i = 0; i < str1.size(); i++)
15     {
16         cout << str1.at(i) << " ";
17     }
18     cout << endl;
19     //修改单个字符
20     str1[0] = 'x';
21     cout << str1 << endl;
22     str1.at(2) = 'c';
23     cout << str1 << endl;
24 }
25 int main()
26 {
27     test01();
28 }

```

- string字符串插入和删除

- ```

1  #include<iostream>
2  using namespace std;
3  //string字符串插入和删除
4  void test01()
5  {
6      string str = "hello";
7      //插入
8      str.insert(1, "4444");//在第一个位置中插入4444
9      cout << str << endl;
10     //删除
11     str.erase(1, 4);//从第一个位置起删除4位
12     cout << str << endl;
13 }
14 int main()
15 {
16     test01();
17 }

```

- 

- string子串

- ```

1  #include<iostream>
2  using namespace std;
3  //string字符串子串
4  void test01()
5  {
6      string str = "hello";
7      string substr = str.substr(1, 3);//从1开始截取，截取三位
8      cout << substr << endl;
9  }
10 void test02()
11 {
12     string email = "zhangsan@163.com";
13     //从邮件中获取用户信息
14     int pos = email.find("@");
15     string userName = email.substr(0, pos);
16     cout << userName << endl;

```



```

17 }
18 int main()
19 {
20     test01();
21     test02();
22 }

```

## vector容器

- vector数据结构和数组非常类似，也称为**单端数组**
- 和普通数组不同的是普通数组是静态空间，而**vector可以动态扩展**
- 动态扩展并不是在原有的空间中接上新空间，而是查找更大的空间，然后将原数据拷贝到新空间，释放原空间
- vector容器的迭代器是支持随机访问的迭代器
- vector的构造函数：创建vector容器

```

1 vector<T>; //采用模板类实现，默认构造函数
2 vector<v.begin(), v.end()>; //将[begin(), end())区间中的元素拷贝给本身
3 vector<n, elem>; //构造函数n个elem拷贝给本身
4 vector<const vector &vec>; //拷贝构造函数

```

```

1 #include<iostream>
2 #include<vector>
3 using namespace std;
4 //vector容器的构造
5 void printVector(vector<int>& v)
6 {
7     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 void test01()
14 {
15     vector<int> v1;
16     for (int i = 0; i < 10; i++)
17     {
18         v1.push_back(i + 1);
19     }
20     printVector(v1);
21     //通过区间方式进行构造
22     vector<int> v2(v1.begin(), v1.end());
23     printVector(v2);
24     //n个elem方式构造
25     vector<int> v3(10, 20);
26     printVector(v3);
27     //拷贝构造
28     vector<int> v4(v3);
29     printVector(v4);
30 }
31 int main()
32 {
33     test01();

```

```
34
35 }
```

- vector赋值操作

```
○ 1 #include<iostream>
2 #include<vector>
3 using namespace std;
4 //vector容器的赋值操作
5 void printVector(vector<int>& v)
6 {
7     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 void test01()
14 {
15     vector<int> v1;
16     for (int i = 0; i < 10; i++)
17     {
18         v1.push_back(i + 1);
19     }
20     //operator
21     vector<int> v2;
22     v2 = v1;
23     //assign
24     vector<int> v3;
25     v3.assign(v1.begin(), v1.end());
26     printVector(v3);
27     //n个elem
28     vector<int> v4;
29     v4.assign(10, 200);
30     printVector(v4);
31
32
33 }
34 int main()
35 {
36     test01();
37
38 }
```

- vector容量和大小

```
○ 1 #include<iostream>
2 #include<vector>
3 using namespace std;
4 //vector容器的容量和大小操作
5 void printVector(vector<int>& v)
6 {
7     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
```

```

12 }
13 void test01()
14 {
15     vector<int> v1;
16     for (int i = 0; i < 10; i++)
17     {
18         v1.push_back(i + 1);
19     }
20     //判断是否为空
21     if (v1.empty())
22     {
23         cout << "v1为空" << endl;
24     }
25     else
26     {
27         cout << "v1不为空" << endl;
28         cout << "v1的容量为" << v1.capacity() << endl;
29         cout << "v1的大小为" << v1.size() << endl;
30     }
31     v1.resize(20, 32); //如果重新指定的大小比原来过长，默认使用0来填充，利用
                        //重载版本可以指定填充值
32     v1.resize(5); //如果重新指定的比原来小了，则会将多余的删除
33     printVector(v1);
34 }
35
36 int main()
37 {
38     test01();
39 }
40 }

```

- vector的插入和删除

```

o 1 #include<iostream>
2 #include<vector>
3 using namespace std;
4 //vector容器的插入和删除
5 void printVector(vector<int>& v)
6 {
7     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 void test01()
14 {
15     vector<int> v1;
16     for (int i = 0; i < 10; i++)
17     {
18         v1.push_back(i + 1); //尾插法插入数据
19     }
20     //尾删法
21     v1.pop_back();
22     printVector(v1);
23     //插入
24     v1.insert(v1.begin()+5, 100); //第一个参数必须是迭代器

```

```

25     printVector(v1);
26     v1.insert(v1.begin(), 2, 34); //插入N个数据
27     printVector(v1);
28     //删除
29     v1.erase(v1.begin()); //参数需要迭代器
30     printVector(v1);
31     v1.erase(v1.begin(), v1.end()); //直接清空容器
32     printVector(v1);
33     v1.clear();
34     printVector(v1); //清空
35 }
36 int main()
37 {
38     test01();
39 }
40 }

```

- vector数据存取

```

○ 1  #include<iostream>
2  #include<vector>
3  using namespace std;
4  //vector容器的数据存取
5  void test01()
6  {
7      vector<int> v1;
8      for (int i = 0; i < 10; i++)
9      {
10         v1.push_back(i + 1); //尾插法插入数据
11     }
12     for (int i = 0; i < v1.size(); i++)
13     {
14         cout << v1[i] << " ";
15     }
16     cout << endl;
17     //利用at方式访问元素
18     for (int i = 0; i < v1.size(); i++)
19     {
20         cout << v1.at(i) << " ";
21     }
22     cout << endl;
23     //获取第一个元素
24     cout << "第一个元素为" << v1.front() << endl;
25     //获取最后一个元素
26     cout << "最后一个元素为" << v1.back() << endl;
27 }
28 int main()
29 {
30     test01();
31 }
32 }

```

- vector互换容器

```

○ 1  #include<iostream>
2  #include<vector>
3  using namespace std;

```

```

4 //vector容器互换
5 void printVector(vector<int>& v)
6 {
7     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 void test01()
14 {
15     vector<int> v1;
16     for (int i = 0; i < 10; i++)
17     {
18         v1.push_back(i + 1); //尾插法插入数据
19     }
20     printVector(v1);
21     vector<int> v2;
22     for (int j = 10; j > 0; j--)
23     {
24         v2.push_back(j);
25     }
26     printVector(v2);
27     cout << "交换后" << endl;
28     v1.swap(v2);
29     printVector(v1);
30     printVector(v2);
31 }
32 //互换的实际用途是收缩储存空间
33 void test02()
34 {
35     vector<int> v3;
36     for (int i = 0; i < 10000; i++)
37     {
38         v3.push_back(i);
39     }
40     cout << "v3的容量为" << v3.capacity() << endl;
41     cout << "v3的大小为" << v3.size() << endl;
42     v3.resize(3); //此时会浪费内存空间
43     cout << "v3的容量为" << v3.capacity() << endl;
44     cout << "v3的大小为" << v3.size() << endl;
45     //巧用swap收缩内存（拷贝构造匿名对象，匿名对象由系统自动回收）
46     vector<int>(v3).swap(v3);
47     cout << "v3的容量为" << v3.capacity() << endl;
48     cout << "v3的大小为" << v3.size() << endl;
49 }
50 }
51 int main()
52 {
53     test01();
54     test02();
55 }

```

- vector预留空间

- 功能：减少vector在动态扩展容量时的扩展次数

- 1 #include<iostream>

```

2  #include<vector>
3  using namespace std;
4  //vector容器预留空间
5  void printVector(vector<int>& v)
6  {
7      for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     vector<int> v1;
16     int num = 0; //统计开辟次数
17     int* p = NULL;
18     for (int i = 0; i < 10; i++)
19     {
20         v1.push_back(i + 1); //尾插法插入数据
21         if (p != &v1[0])
22         {
23             p = &v1[0];
24             num++; //相当于开辟了多次内存
25         }
26     }
27     cout << num << endl;
28 }
29
30 void test02()
31 {
32     vector<int> v1;
33     int num = 0; //统计开辟次数
34     v1.reserve(10); //预留数据空间
35     int* p = NULL;
36     for (int i = 0; i < 10; i++)
37     {
38         v1.push_back(i + 1); //尾插法插入数据
39         if (p != &v1[0])
40         {
41             p = &v1[0];
42             num++; //相当于开辟了多次内存
43         }
44     }
45     cout << num << endl;
46 }
47
48 int main()
49 {
50     test01();
51     test02();
52 }

```

o

## deque容器

- deque容器是双端数组，可以对头端进行插入和删除操作

- deque和vector的区别
  - vector对于头部的插入删除效率低，数据量越大，效率越低
  - deque相对而言对头部的插入和删除速度比vector快
  - vector访问元素时的速度会比deque快，这与两者内部实现有关
- deque内部有中控器维护每一段缓冲区的内容，缓冲区中存放真实数据；中控器维护的是每个缓冲区的地址，使得deque像一片连续的内存空间
- deque容器的迭代器是支持随机访问的
- deque的构造函数

```

1  #include<iostream>
2  #include<deque>
3  using namespace std;
4  //deque容器的构造函数
5  void printDeque(const deque<int>& d1)
6  {
7      for (deque<int>::const_iterator it = d1.begin(); it !=
8          d1.end(); it++)//const_iterator为只读迭代器
9      {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14 void test01()
15 {
16     deque<int> d1;
17     for (int i = 0; i < 10; i++)
18     {
19         d1.push_back(i);
20     }
21     printDeque(d1);
22     deque<int> d2(d1.begin(), d1.end());//区间方式赋值
23     printDeque(d2);
24     deque<int> d3(10, 100);//n个elem
25     printDeque(d3);
26     deque<int> d4(d1);//拷贝构造
27     printDeque(d4);
28 }
29 int main()
30 {
31     test01();
32 }

```

- deque的赋值操作

```

1  #include<iostream>
2  #include<deque>
3  using namespace std;
4  //deque容器的赋值操作
5  void printDeque(const deque<int>& d1)
6  {
7      for (deque<int>::const_iterator it = d1.begin(); it !=
8          d1.end(); it++)//const_iterator为只读迭代器
9      {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }

```

```

12 }
13 void test01()
14 {
15     deque<int> d1;
16     for (int i = 0; i < 10; i++)
17     {
18         d1.push_back(i + 1);
19     }
20     printDeque(d1);
21     //operator赋值
22     deque<int> d2;
23     d2 = d1;
24     printDeque(d2);
25     //assign赋值
26     deque<int> d3;
27     d3.assign(d1.begin(), d1.end());
28     printDeque(d3);
29     deque<int> d4;
30     d4.assign(10, 100);
31     printDeque(d4);
32 }
33 int main()
34 {
35     test01();
36 }

```

- deque大小操作 (deque容器没有容量这个概念, 可以无限扩展)

```

o 1 #include<iostream>
2 #include<deque>
3 using namespace std;
4 //deque容器的大小操作
5 void printDeque(const deque<int>& d1)
6 {
7     for (deque<int>::const_iterator it = d1.begin(); it !=
8         d1.end(); it++)//const_iterator为只读迭代器
9     {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14 void test01()
15 {
16     deque<int> d1;
17     for (int i = 0; i < 10; i++)
18     {
19         d1.push_back(i + 1);
20     }
21     printDeque(d1);
22     if (d1.empty())
23     {
24         cout << "d1为空" << endl;
25     }
26     else
27     {
28         cout << "d1不为空" << endl;
29         cout << "d1的大小为" << d1.size() << endl;
30     }
31 }

```



```

29     }
30     d1.resize(15);
31     printDeque(d1);
32     d1.resize(15, 23);
33     printDeque(d1);
34     d1.resize(5);
35     printDeque(d1);
36
37 }
38 int main()
39 {
40     test01();
41 }

```

- deque的插入和删除

```

o 1  #include<iostream>
2  #include<deque>
3  using namespace std;
4  //deque容器的插入和删除操作
5  void printDeque(const deque<int>& d1)
6  {
7      for (deque<int>::const_iterator it = d1.begin(); it !=
d1.end(); it++)//const_iterator为只读迭代器
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 //两端操作
14 void test01()
15 {
16     deque<int> d1;
17     //尾插
18     d1.push_back(20);
19     d1.push_back(30);
20     //头插
21     d1.push_front(60);
22     d1.push_front(90);
23     printDeque(d1);
24     //尾删
25     d1.pop_back();
26     //头删
27     d1.pop_front();
28     printDeque(d1);
29 }
30 void test02()
31 {
32     deque<int> d2;
33     //尾插
34     d2.push_back(20);
35     d2.push_back(30);
36     d2.push_front(60);
37     d2.push_front(90);
38
39     //insert插入
40     d2.insert(d2.begin(), 34);

```

```

41     printDeque(d2);
42     d2.insert(d2.begin(), 3, 200); //插入3个200
43     printDeque(d2);
44     //按照区间插入数据
45     deque<int> d3;
46     d3.push_back(10);
47     d3.push_back(20);
48     d3.push_back(30);
49     d3.push_back(40);
50     d2.insert(d2.begin(), d3.begin(), d3.end());
51     printDeque(d2);
52 }
53 void test03()
54 {
55     deque<int> d1;
56     d1.push_back(10);
57     d1.push_back(20);
58     d1.push_back(30);
59     d1.push_back(40);
60     //删除操作
61     deque<int>::iterator it = d1.begin();
62     it++;
63     d1.erase(it); //删除第it个数据
64     printDeque(d1);
65     d1.erase(d1.begin(), d1.end()); //区间删除
66     d1.clear();
67 }
68 int main()
69 {
70     test01();
71     test02();
72     test03();
73 }

```

- deque的数据存取

```

○ 1  #include<iostream>
2  #include<deque>
3  using namespace std;
4  //deque容器的数据存取
5  void test01()
6  {
7      deque<int> d1;
8      d1.push_back(20);
9      d1.push_back(30);
10     d1.push_front(60);
11     d1.push_front(90);
12     d1.push_front(100);
13     //通过[]方式来访问元素
14     for (int i = 0; i < d1.size(); i++)
15     {
16         cout << d1[i] << " ";
17     }
18     cout << endl;
19     //通过at方式来访问
20     for (int i = 0; i < d1.size(); i++)
21     {

```

```

22     cout << d1.at(i) << " ";
23 }
24 cout << endl;
25 //访问头部和尾部元素
26 cout << "d1的头部元素为" << d1.front() << endl;
27 cout << "d1的尾部元素为" << d1.back() << endl;
28 }
29
30 int main()
31 {
32     test01();
33 }

```

- deque排序

```

o 1 #include<iostream>
2 #include<deque>
3 #include<algorithm>
4 using namespace std;
5 //deque容器的数据存取
6 void printDeque(const deque<int>& d1)
7 {
8     for (deque<int>::const_iterator it = d1.begin(); it !=
d1.end(); it++)//const_iterator为只读迭代器
9     {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14 //两端操作
15 void test01()
16 {
17     deque<int> d1;
18     d1.push_back(20);
19     d1.push_back(650);
20     d1.push_front(50);
21     d1.push_front(90);
22     d1.push_front(10);
23     printDeque(d1);
24     //排序
25     //对于支持随机访问的迭代器的容器都可以利用sort算法直接对其进行排序
26     sort(d1.begin(), d1.end());
27     cout << "排序后: " << endl;
28     printDeque(d1);
29 }
30
31 int main()
32 {
33     test01();
34 }
35 }

```

- 容器案例：评委打分

```

1 #include<vector>

```

```

2  #include<iostream>
3  #include<algorithm>
4  #include<deque>
5  #include<ctime>
6  using namespace std;
7
8  //选手类
9  class Person
10 {
11 public:
12     Person(string name, int score)
13     {
14         this->m_Name = name;
15         this->m_Score = score;
16     }
17     string m_Name;
18     int m_Score;
19 };
20 void creatPerson(vector<Person>& v)
21 {
22     string nameSeed = "ABCDE";
23     for (int i = 0; i < 5; i++)
24     {
25         string name = "选手";
26         name += nameSeed[i];
27         int score = 0;
28         Person p(name, score);
29         v.push_back(p);
30     }
31 }
32 void setScore(vector<Person> v)
33 {
34     for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
35     {
36         deque<int> d;
37         for (int i = 0; i < 10; i++)
38         {
39             int score = rand() % 41 + 60;
40             d.push_back(score);
41         }
42         cout << "选手" << it->m_Name << "\t" << "打分: " << endl;
43         for (deque<int>::iterator dit = d.begin(); dit != d.end();
dit++)
44         {
45             cout << *dit << "\t";
46         }
47         cout << endl;
48         sort(d.begin(), d.end());
49         d.pop_front();
50         d.pop_back(); //去除最高分和最低分
51         //取平均分
52         int sum = 0;
53         for (deque<int>::iterator dit = d.begin(); dit != d.end();
dit++)
54         {
55             sum += *dit;
56         }
57         int avg = sum / d.size();

```

```

58         //cout << avg << endl;
59         it->m_Score = avg; //将平均分赋值给选手
60         //cout << it->m_Score << endl;
61     }
62     cout << "*****" << endl;
63 }
64 void showScore(vector<Person>& v)
65 {
66     for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
67     {
68         cout << "姓名: " << it->m_Name << "\t分数: " << it->m_Score <<
endl;
69     }
70 }
71 int main()
72 {
73     //随机数种子
74     srand((unsigned int)time(NULL));
75     //创建五个选手
76     vector<Person> v;
77     creatPerson(v);
78     //给五个选手打分
79     setScore(v);
80     //显示分数
81     showScore(v);
82 }

```

## stack容器

- stack容器是一种**先进后出**的数据结构，只有一个出口（栈）
- 栈不允许有遍历行为

- ```

1  #include<stack>
2  #include<iostream>
3  using namespace std;
4  //栈（stack）容器
5  void test01()
6  {
7      stack<int> s;
8      //入栈
9      for (int i = 0; i < 5; i++)
10     {
11         s.push(i);
12     }
13     cout << "栈的大小为" << s.size() << endl;
14     //只要栈不为空，查看栈顶，并执行出栈操作
15     while (!s.empty())
16     {
17         cout << "栈顶元素为" << s.top() << endl;
18         s.pop();
19     }
20     cout << "栈的大小为" << s.size() << endl;
21 }
22
23 int main()
24 {
25     test01();

```

## queue容器

- 队列是一种**先进先出**的数据结构，有两个出口
- 对头出队，对尾入队
- 只有队头和队尾元素能被外界访问，因此不允许有遍历行为

```

1  #include<queue>
2  #include<iostream>
3  using namespace std;
4  //队列容器
5  class Person
6  {
7  public:
8      Person(string name, int age)
9      {
10         this->m_Name = name;
11         this->m_Age = age;
12     }
13     string m_Name;
14     int m_Age;
15 };
16 void test01()
17 {
18     queue<Person> q;
19     Person p1("关羽", 43);
20     Person p2("张飞", 32);
21     Person p3("刘备", 54);
22     Person p4("董卓", 76);
23     //入队
24     q.push(p1);
25     q.push(p2);
26     q.push(p3);
27     q.push(p4);
28     while (!q.empty())
29     {
30         //查看队头
31         cout << "队头元素--姓名" << q.front().m_Name << "\t年龄" <<
q.front().m_Age << endl;
32         //查看队尾
33         cout << "队尾元素--姓名" << q.back().m_Name << "\t年龄" <<
q.back().m_Age << endl;
34         //出队
35         q.pop();
36     }
37
38
39 }
40 int main()
41 {
42     test01();
43 }

```

## list容器

- 功能：将数据进行**链式存储**
- 链表是一种物理储存单元上非连续的储存结构，数据元素的逻辑顺序是通过链表中的指针链接来实现的
- 链表由一系列结点组成；结点由储存数据元素的**数据域**和储存下一个结点地址的**指针域**构成
- STL中的链表是一个**双向循环链表**
- 链表可以对任意的位置进行快速的添加，插入或者删除元素
- 链表的迭代器只支持前移和后移，不支持随机访问，属于**双向迭代器**
- 链表采用动态储存分配，不会造成内存的浪费
- 链表的插入和删除操作不会使原有的迭代器失效（vector容器不成立）
- list的构造函数

```

○ 1  #include<list>
2  #include<iostream>
3  using namespace std;
4  //list容器的构造函数
5  void printList(list<int>& L)
6  {
7      for (list<int>::iterator it = L.begin(); it != L.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     list<int> l1;
16     for (int i = 0; i < 5; i++)
17     {
18         //默认构造
19         l1.push_back(i);
20     }
21     printList(l1);
22     //区间方式构造
23     list<int> l2(l1.begin(), l1.end());
24     printList(l2);
25     //拷贝构造
26     list<int> l3(l2);
27     printList(l3);
28     //n个elem
29     list<int> l4(10, 233);
30     printList(l4);
31 }
32 int main()
33 {
34     test01();
35 }

```

- list容器的赋值和交换

```

○ 1  #include<list>
2  #include<iostream>
3  using namespace std;
4  //list容器赋值和交换
5  void printList(list<int>& L)

```

```

6  {
7      for (list<int>::iterator it = L.begin(); it != L.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     list<int> L1;
16     for (int i = 0; i < 5; i++)
17     {
18         //默认构造
19         L1.push_back(i);
20     }
21     //operator
22     list<int> L2;
23     L2 = L1;
24     printList(L2);
25     //assign赋值
26     list<int> L3;
27     L3.assign(L2.begin(), L2.end());
28     printList(L3);
29     list<int> L4;
30     L4.assign(5, 233);
31     printList(L4);
32 }
33 void test02()
34 {
35     list<int> L1;
36     list<int> L2;
37     for (int i = 0; i < 5; i++)
38     {
39         //默认构造
40         L1.push_back(i);
41         L2.push_back(i + 10);
42     }
43     cout << "交换前:" << endl;
44     printList(L1);
45     printList(L2);
46     L1.swap(L2);
47     cout << "交换后: " << endl;
48     printList(L1);
49     printList(L2);
50 }
51 int main()
52 {
53     test01();
54     test02();
55 }

```

- list容器大小操作

- ```

1  #include<list>
2  #include<iostream>
3  using namespace std;
4  //list容器的大小操作

```



```

5 void printList(list<int>& L)
6 {
7     for (list<int>::iterator it = L.begin(); it != L.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 void test01()
14 {
15     list<int> L1;
16     for (int i = 0; i < 5; i++)
17     {
18         //默认构造
19         L1.push_back(i);
20     }
21     //判断容器是否为空
22     if (L1.empty())
23     {
24         cout << "L1为空" << endl;
25     }
26     else
27     {
28         cout << "L1的元素个数为" << L1.size() << endl;
29     }
30     //重新指定大小
31     L1.resize(10);
32     printList(L1);
33     L1.resize(2);
34     printList(L1);
35 }
36 }
37 int main()
38 {
39     test01();
40 }

```

- list插入和删除

```

1  #include<list>
2  #include<iostream>
3  using namespace std;
4  //list容器的插入和删除操作
5  void printList(list<int>& L)
6  {
7      for (list<int>::iterator it = L.begin(); it != L.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     list<int> L1;
16     for (int i = 0; i < 5; i++)
17     {
18         //默认构造

```

```

19     L1.push_back(i);
20     L1.push_front(i + 4);
21 }
22 printList(L1);
23 //尾删
24 L1.pop_back();
25 //头删
26 L1.pop_front();
27 //insert插入
28 L1.insert(L1.begin(), 20);
29 list<int>::iterator it = L1.begin();
30 L1.insert(++it, 32);
31 printList(L1);
32 //删除
33 it = L1.begin();
34 L1.erase(it++); //参数一定是迭代器
35 printList(L1);
36 //移除
37 L1.push_back(100);
38 L1.push_back(100);
39 L1.push_back(100);
40 printList(L1);
41 L1.remove(100);
42 printList(L1);
43 //清空
44 L1.clear();
45 }
46 int main()
47 {
48     test01();
49 }

```

- list的数据存取

```

o 1 #include<list>
2 #include<iostream>
3 using namespace std;
4 //list容器的数据存取
5 void printList(list<int>& L)
6 {
7     for (list<int>::iterator it = L.begin(); it != L.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 void test01()
14 {
15     list<int> L1;
16     for (int i = 0; i < 5; i++)
17     {
18         //默认构造
19         L1.push_back(i);
20         L1.push_front(i + 4);
21     }
22     cout << "list的尾部为: " << L1.back() << endl;
23     cout << "list的头部为" << L1.front() << endl;

```

```

24     //list的本质是一个链表，不是线性空间存储数据，迭代器不支持随机访问（只支持
    ++或者--操作）
25 }
26 int main()
27 {
28     test01();
29 }

```

- list的反转和排序

```

○ 1 #include<list>
2 #include<iostream>
3 using namespace std;
4 //list容器的反转和排序
5 void printList(list<int>& L)
6 {
7     for (list<int>::iterator it = L.begin(); it != L.end(); it++)
8     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 bool myCompare(int v1, int v2)//降序排序
14 {
15
16     return v1 > v2;
17 }
18 void test01()
19 {
20     list<int> L1;
21     for (int i = 0; i < 5; i++)
22     {
23         //默认构造
24         L1.push_back(i);
25         L1.push_front(i + 4);
26     }
27     printList(L1);
28     L1.reverse();
29     cout << "反转后: " << endl;
30     printList(L1);
31     L1.sort();
32     cout << "排序后: " << endl;
33     printList(L1);
34     cout << "降序排序后" << endl;
35     L1.sort(myCompare);
36     printList(L1);
37
38 }
39 int main()
40 {
41     test01();
42 }

```

- 所有不支持随机访问迭代器的容器不可以使用标准算法；不支持随机访问迭代器的容器内部会提供对应的算法
- 自定义数据高级排序

■

```
1  #include<list>
2  #include<iostream>
3  using namespace std;
4  //list容器的排序案例
5  class Person
6  {
7  public:
8      Person(string name, int age, int height)
9      {
10         this->m_age = age;
11         this->m_name = name;
12         this->m_height = height;
13     }
14     string m_name;
15     int m_age;
16     int m_height;
17
18 private:
19
20 };
21 //指定排序规则
22 bool myCompareAge(Person& p1, Person &p2)
23 {
24     if (p1.m_age == p2.m_age)
25     {
26         return p1.m_height > p2.m_height;
27     }
28     else
29     {
30         return p1.m_age > p2.m_age;
31     }
32 }
33
34 void printList(list<Person>& L)
35 {
36     for (list<Person>::iterator it = L.begin(); it != L.end();
37         it++)
38     {
39         cout << "姓名:" << (*it).m_name << "\t年龄:" <<
40             (*it).m_age
41             << "\t身高:" << (*it).m_height << endl;
42     }
43 }
44 void test01()
45 {
46     Person p1("关羽", 43, 187);
47     Person p2("张飞", 54, 176);
48     Person p3("刘备", 54, 175);
49     Person p4("董卓", 76, 165);
50     list<Person> L;
51     L.push_back(p1);
52     L.push_back(p2);
53     L.push_back(p3);
54     L.push_back(p4);
55     printList(L);
56     cout << "-----"
57     " << endl;
```

```

55     cout << "按照年龄排序" << endl;
56     L.sort(myCompareAge);
57     printList(L);
58 }
59 int main()
60 {
61     test01();
62 }

```

## set/multiset容器

- 所有元素都会在插入时**自动排序**
- set/multiset属于关联式容器，底层结构是**二叉树**
- set/multiset容器的区别：set不允许有重复的数据；multiset允许有重复的数据
- set的构造和赋值

```

o 1  #include<set>
2  #include<iostream>
3  using namespace std;
4  //set容器
5  void printSet(set<int>& s)
6  {
7      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     set<int> s1;
16     s1.insert(1); //插入数据只有insert函数
17     s1.insert(7);
18     s1.insert(2); //set不允许插入重复的值
19     s1.insert(2);
20     printSet(s1);
21     //拷贝构造
22     set<int> s2(s1);
23     printSet(s2);
24     //赋值操作
25     set<int> s3;
26     s3 = s2;
27     printSet(s3);
28 }
29 int main()
30 {
31     test01();
32 }

```

- set的大小和交换

```

o 1  #include<set>
2  #include<iostream>
3  using namespace std;
4  //set容器的大小和交换
5  void printSet(set<int>& s)

```

```

6  {
7      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     set<int> s1;
16     s1.insert(1); //插入数据只有insert函数
17     s1.insert(7);
18     s1.insert(2); //set不允许插入重复的值
19     s1.insert(2);
20     printSet(s1);
21     //判断是否为空
22     if (s1.empty())
23     {
24         cout << "s1为空" << endl;
25     }
26     else
27     {
28         cout << "s1的大小为" << s1.size() << endl;
29     }
30     set<int> s2;
31     s2.insert(43);
32     s2.insert(21);
33     s2.insert(53);
34     s2.insert(57);
35     s2.swap(s1); //交换容器
36     printSet(s1);
37     printSet(s2);
38
39
40 }
41 int main()
42 {
43     test01();
44 }

```

- set的插入和删除

- ```

1  #include<set>
2  #include<iostream>
3  using namespace std;
4  //set容器的插入和删除
5  void printSet(set<int>& s)
6  {
7      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     set<int> s2;

```

```

16     s2.insert(43);
17     s2.insert(21);
18     s2.insert(53);
19     s2.insert(57);
20     s2.erase(21); //相当于list的remove
21     printSet(s2);
22     s2.erase(s2.begin());
23     //清空
24     s2.erase(s2.begin(), s2.end());
25     s2.clear();
26 }
27 int main()
28 {
29     test01();
30 }

```

- set的查找和统计

- ```

1  #include<set>
2  #include<iostream>
3  using namespace std;
4  //set容器的插入和删除
5  void printSet(set<int>& s)
6  {
7      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     set<int> s2;
16     s2.insert(43);
17     s2.insert(21);
18     s2.insert(53);
19     s2.insert(57);
20     //查找
21     set<int>::iterator pos = s2.find(21); //find函数返回end值
22     if (pos != s2.end())
23     {
24         cout << "找到元素" << *pos << endl;
25     }
26     else
27     {
28         cout << "未找到该元素" << endl;
29     }
30 }
31 void test02() //统计
32 {
33     set<int> s2;
34     s2.insert(43);
35     s2.insert(21);
36     s2.insert(53);
37     s2.insert(57);
38     //统计元素的个数
39     int num = s2.count(21); //对于set而言, count返回值要么是0, 要么是1

```

```

40     cout << num << endl;
41 }
42 int main()
43 {
44     test01();
45     test02();
46 }

```

- set和multiset的区别

```

o 1  #include<set>
2  #include<iostream>
3  using namespace std;
4  void printSet(set<int>& s)
5  {
6      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
7      {
8          cout << *it << " ";
9      }
10     cout << endl;
11 }
12 void test01()
13 {
14     set<int> s2;
15     s2.insert(43);
16     s2.insert(21);
17     s2.insert(53);
18     s2.insert(57);
19     pair<set<int>::iterator, bool> ret = s2.insert(20);
20     if (ret.second)
21     {
22         cout << "第一次插入成功" << endl;
23     }
24     else
25     {
26         cout << "第一次插入失败" << endl;
27     }
28     ret = s2.insert(20);
29     if (ret.second)
30     {
31         cout << "第二次插入成功" << endl;
32     }
33     else
34     {
35         cout << "第二次插入失败" << endl;
36     }
37     multiset<int> s1;
38     //允许插入重复值
39     s1.insert(10);
40     s1.insert(10);
41     for (multiset<int>::iterator it = s1.begin(); it != s1.end();
42 it++)
43     {
44         cout << *it << "\t";
45     }
46     cout << endl;
47 }

```



```

47 void test02()//统计
48 {
49     set<int> s2;
50     s2.insert(43);
51     s2.insert(21);
52     s2.insert(53);
53     s2.insert(57);
54     //统计元素的个数
55     int num = s2.count(21);//对于set而言，count返回值要么是0，要么是1
56     cout << num << endl;
57 }
58 int main()
59 {
60     test01();
61 }

```

- pair队组创建

- ```

1  #include<set>
2  #include<iostream>
3  using namespace std;
4  //pair队组的创建
5  void printSet(set<int>& s)
6  {
7      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
8      {
9          cout << *it << " ";
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     pair<string, int> p1("tom", 32);
16     cout << p1.first << endl;
17     cout << p1.second << endl;
18 }
19
20 int main()
21 {
22     test01();
23 }

```

- set容器排序

- ```

1  #include<set>
2  #include<iostream>
3  using namespace std;
4  //set容器的排序
5  //仿函数:本质上是一个类型
6  class MyCompare
7  {
8  public:
9      bool operator()(int v1, int v2) const
10     {
11         return v1 > v2;
12     }
13 };

```

```

14 void printSet(set<int>& s)
15 {
16     for (set<int>::iterator it = s.begin(); it != s.end(); it++)
17     {
18         cout << *it << " ";
19     }
20     cout << endl;
21 }
22 void test01()
23 {
24     set<int> s1;
25     s1.insert(20);
26     s1.insert(30);
27     s1.insert(40);
28     s1.insert(50);
29     s1.insert(60);
30     printSet(s1);
31     //指定排序规则为从大到小，必须在数据未插入之前改变规则
32     set<int, MyCompare> s2;
33     s2.insert(20);
34     s2.insert(30);
35     s2.insert(40);
36     s2.insert(50);
37     s2.insert(60);
38     for (set<int, MyCompare>::iterator it = s2.begin(); it !=
s2.end(); it++)
39     {
40         cout << *it << endl;
41     }
42 }
43 }
44
45 int main()
46 {
47     test01();
48 }

```

o set容器自定义数据如何进行排序

```

1  #include<set>
2  #include<iostream>
3  using namespace std;
4  //set容器的排序
5  //仿函数:本质上是一个类型
6  class Person
7  {
8  public:
9      Person(string name, int age)
10     {
11         this->m_name = name;
12         this->m_age = age;
13     }
14     string m_name;
15     int m_age;
16 };
17 class comparePerson
18 {

```

```

19 public:
20     bool operator()(const Person &p1, const Person &p2) const
21     {
22         return p1.m_age > p2.m_age;
23     }
24 };
25
26 void printSet(set<int>& s)
27 {
28     for (set<int>::iterator it = s.begin(); it != s.end();
29         it++)
30     {
31         cout << *it << " ";
32     }
33     cout << endl;
34 }
35 void test01()
36 {
37     //自定义的数据类型需要指定排序规则
38     set<Person, comparePerson> s1;
39     Person p1("关羽", 43);
40     Person p2("张飞", 32);
41     Person p3("刘备", 54);
42     Person p4("董卓", 76);
43
44     s1.insert(p1);
45     s1.insert(p2);
46     s1.insert(p3);
47     s1.insert(p4);
48     for (set<Person, comparePerson>::iterator it = s1.begin();
49         it != s1.end(); it++)
50     {
51         cout << "姓名" << (*it).m_name << "\t年龄" <<
52         (*it).m_age << endl;
53     }
54 }
55 int main()
56 {
57     test01();
58 }

```

## map/multimap容器

- map容器所有元素都是pair
- pair中第一个元素为key(键值)，起到索引作用，第二个元素为value (实值)
- 所有的元素都会根据元素的键值进行自动排序
- map/multimap属于关联式容器，底层结构是用**二叉树**实现
- map/multimap的区别：map不允许有重复的key值元素；multimap允许有重复的key值
- map构造和赋值

```

1 #include<map>
2 #include<iostream>
3 using namespace std;
4 //map容器的构造和赋值
5 void printMap(map<int, int>& m)

```

```

6 {
7     for (map<int, int>::iterator it = m.begin(); it != m.end();
it++)
8     {
9         cout << "key = " << (*it).first << "\tvalue = " <<
(*it).second << endl;
10    }
11 }
12 void test01()
13 {
14     //创建map容器
15     //默认构造
16     map<int, int> m;
17     //map容器中元素都是成对出现，插入数值时需要使用pair
18     m.insert(pair<int, int>(1, 20));
19     m.insert(pair<int, int>(2, 30));
20     m.insert(pair<int, int>(3, 60));
21     m.insert(pair<int, int>(4, 40));
22     m.insert(pair<int, int>(5, 90));
23     printMap(m);
24     //拷贝构造
25     map<int, int> m2(m);
26     printMap(m2);
27     //赋值
28     map<int, int> m3;
29     m3 = m2;
30     printMap(m3);
31 }
32 }
33 int main()
34 {
35     test01();
36 }

```

- map的大小和交换

```

o 1 #include<map>
2 #include<iostream>
3 using namespace std;
4 //map容器的构造和赋值
5 void printMap(map<int, int>& m)
6 {
7     for (map<int, int>::iterator it = m.begin(); it != m.end();
it++)
8     {
9         cout << "key = " << (*it).first << "\tvalue = " <<
(*it).second << endl;
10    }
11    cout << endl;
12 }
13 void test01()
14 {
15     //创建map容器
16     //默认构造
17     map<int, int> m;
18     m.insert(pair<int, int>(1, 20));
19     m.insert(pair<int, int>(2, 30));

```

```

20     m.insert(pair<int, int>(3, 60));
21     m.insert(pair<int, int>(4, 40));
22     m.insert(pair<int, int>(5, 90));
23     printMap(m);
24     //判断是否为空
25     if (m.empty())
26     {
27         cout << "map为空" << endl;
28     }
29     else
30     {
31         cout << "*****" << endl;
32
33         cout << m.size() << endl;
34     }
35 }
36 void test02()
37 {
38     //交换
39     map<int, int> m1;
40     m1.insert(pair<int, int>(1, 20));
41     m1.insert(pair<int, int>(2, 30));
42     m1.insert(pair<int, int>(3, 60));
43     m1.insert(pair<int, int>(4, 40));
44     m1.insert(pair<int, int>(5, 90));
45     map<int, int> m2;
46     m2.insert(pair<int, int>(1, 43));
47     m2.insert(pair<int, int>(2, 35));
48     m2.insert(pair<int, int>(3, 54));
49     m2.insert(pair<int, int>(4, 63));
50     m2.insert(pair<int, int>(5, 24));
51     cout << "交换前" << endl;
52     printMap(m1);
53     printMap(m2);
54     cout << "交换后" << endl;
55     m2.swap(m1);
56     printMap(m1);
57     printMap(m2);
58 }
59 }
60 int main()
61 {
62     test01();
63     test02();
64 }

```

- map元素的插入和删除

- ```

1  #include<map>
2  #include<iostream>
3  using namespace std;
4  //map容器的插入和删除
5  void printMap(map<int, int>& m)
6  {
7      for (map<int, int>::iterator it = m.begin(); it != m.end();
it++)
8      {

```

```

9         cout << "key = " << (*it).first << "\tvalue = " <<
(*it).second << endl;
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     //创建map容器
16     //默认构造
17     map<int, int> m;
18     m.insert(pair<int, int>(1, 20));
19     m.insert(pair<int, int>(2, 30));
20     m.insert(pair<int, int>(3, 60));
21     m.insert(pair<int, int>(4, 40));
22     m.insert(pair<int, int>(5, 90));
23     printMap(m);
24     //判断是否为空
25     if (m.empty())
26     {
27         cout << "map为空" << endl;
28     }
29     else
30     {
31         //插入
32         m.insert(pair<int, int>(6, 32));
33         printMap(m);
34         m.insert(make_pair(8, 32));
35         printMap(m);
36         m.insert(map<int, int>::value_type(9, 23));
37         printMap(m);
38         m[8] = 43; //重载[], 不建议使用该方式
39         cout << m[3] << endl; //通常使用key值来访问value, 而不是用来插入
40         printMap(m);
41         //删除
42         m.erase(m.begin());
43         printMap(m);
44         m.erase(2); //按照key删除
45         printMap(m);
46         m.erase(m.begin(), m.end()); //区间方式删除
47         printMap(m);
48         //清空
49         m.clear();
50
51     }
52 }
53 int main()
54 {
55     test01();
56 }

```

- map的查找和统计

- ```

1 #include<map>
2 #include<iostream>
3 using namespace std;
4 //map容器的查找和统计
5 void printMap(map<int, int>& m)

```

```

6  {
7      for (map<int, int>::iterator it = m.begin(); it != m.end();
it++)
8      {
9          cout << "key = " << (*it).first << "\tvalue = " <<
(*it).second << endl;
10     }
11     cout << endl;
12 }
13 void test01()
14 {
15     //创建map容器
16     //默认构造
17     map<int, int> m;
18     m.insert(pair<int, int>(1, 20));
19     m.insert(pair<int, int>(2, 30));
20     m.insert(pair<int, int>(3, 60));
21     m.insert(pair<int, int>(4, 40));
22     m.insert(pair<int, int>(5, 90));
23     printMap(m);
24     //判断是否为空
25     if (m.empty())
26     {
27         cout << "map为空" << endl;
28     }
29     else
30     {
31         //查找
32         map<int, int>::iterator pos = m.find(3); //返回值为迭代器
33         if (pos != m.end())
34         {
35             cout << "查找到了元素 key = " << (*pos).first << "\tvalue
= " << (*pos).second << endl;
36         }
37         else
38         {
39             cout << "未找到元素" << endl;
40         }
41         //统计
42         int num = m.count(3); //输出要么是0, 要么是1; multimap的值可以大于
1
43         cout << num << endl;
44     }
45 }
46
47 int main()
48 {
49     test01();
50 }

```

- map容器排序

- ```

1  #include<map>
2  #include<iostream>
3  using namespace std;
4  //map容器的排序
5  class myCompare

```

```

6  {
7  public:
8      bool operator()(int v1, int v2) const
9      {
10         return v1 > v2;
11     }
12 };
13 void printMap(map<int, int, myCompare>& m)
14 {
15     for (map<int, int>::iterator it = m.begin(); it != m.end();
16         it++)
17     {
18         cout << "key = " << (*it).first << "\tvalue = " <<
19         (*it).second << endl;
20     }
21     cout << endl;
22 }
23 void test01()
24 {
25     //创建map容器
26     //默认构造
27     map<int, int, myCompare> m;
28     m.insert(pair<int, int>(1, 20));
29     m.insert(pair<int, int>(2, 30));
30     m.insert(pair<int, int>(3, 60));
31     m.insert(pair<int, int>(4, 40));
32     m.insert(pair<int, int>(5, 90));
33     printMap(m);
34     //判断是否为空
35     if (m.empty())
36     {
37         cout << "map为空" << endl;
38     }
39     else
40     {
41         printMap(m);
42     }
43 }
44 int main()
45 {
46     test01();
47 }

```

- map案例

```

○ 1  #include<iostream>
2  #include<map>
3  #include<vector>
4  #include<time.h>
5  using namespace std;
6  #define CEHUA 1
7  #define MEISHU 2
8  #define YANFA 3
9  class worker
10 {
11 public:

```



```

12
13     string m_name;
14     int m_salary;
15 };
16 void creatworker(vector<worker> &w)
17 {
18     string nameSeed = "ABCDEFGHJIJ";
19     for (int i = 0; i < 10; i++)
20     {
21         worker worker;
22         worker.m_name = "员工";
23         worker.m_name += nameSeed[i];
24         worker.m_salary = rand() % 10000 + 10000;
25         w.push_back(worker);
26     }
27 }
28
29 void setGroup(vector<Worker> &v, multimap<int, worker>& g)
30 {
31     for (vector<worker>::iterator it = v.begin(); it != v.end();
32 it++)
33     {
34         int deptId = rand() % 3 + 1;
35         g.insert(make_pair(deptId, *it));
36     }
37 }
38 void showWorkerByGroup(multimap<int, worker>& g)
39 {
40     cout << "策划部门" << endl;
41     multimap<int, worker>::iterator pos = g.find(CEHUA);
42     int count = g.count(CEHUA);
43     int index = 0;
44     for (; pos != g.end() && index < count; pos++, index++)
45     {
46         cout << "姓名: " << pos->second.m_name << "\t工资: " << pos-
47 >second.m_salary << endl;
48     }
49     cout << "-----" << endl;
50     cout << "美术部门" << endl;
51     pos = g.find(MEISHU);
52     count = g.count(MEISHU);
53     index = 0;
54     for (; pos != g.end() && index < count; pos++, index++)
55     {
56         cout << "姓名: " << pos->second.m_name << "\t工资: " << pos-
57 >second.m_salary << endl;
58     }
59     cout << "-----" << endl;
60     cout << "研发部门" << endl;
61     pos = g.find(YANFA);
62     count = g.count(MEISHU);
63     index = 0;
64     for (; pos != g.end() && index < count; pos++, index++)
65     {
66         cout << "姓名: " << pos->second.m_name << "\t工资: " << pos-
67 >second.m_salary << endl;
68     }
69 }

```

```

66 int main()
67 {
68     srand((unsigned int)time(NULL));
69     //创建员工
70     vector<Worker> v_worker;
71     creatWorker(v_worker);
72     //员工分组
73     multimap<int, Worker> g_worker;
74     setGroup(v_worker, g_worker);
75     //分组显示员工
76     showWorkerByGroup(g_worker);
77 }

```

## 函数对象（仿函数）

- 重载函数调用操作符的类，其对象称为函数对象
- 函数对象使用重载的（）时，行为类似函数调用，也叫仿函数
- 本质：**函数对象是一个类而不是一个函数**
- 函数对象的使用
  - 函数对象在使用时，可以像普通函数那样调用，可以有参数，可以有返回值
  - 函数对象超出普通函数的概念，函数对象可以有自己的状态
  - 函数对象可以作为参数传递

- ```

1  #include<iostream>
2  using namespace std;
3  //函数对象
4  class MyAdd
5  {
6  public:
7      int operator()(int v1, int v2)
8      {
9          return v1 + v2;
10     }
11
12     private:
13
14 };
15 //函数对象可以有自己的状态
16 class MyPrint
17 {
18 public:
19     void operator()(string test)
20     {
21         cout << test << endl;
22         this->count++;
23     }
24     int count = 0; //内部自己的状态记录
25 };
26
27 void test01()
28 {
29     MyAdd myadd;
30     cout << myadd(10, 10) << endl;
31 }
32 void test02()
            
```

```

33 {
34     MyPrint myprint;
35     myprint("liujun");
36     cout << myprint.count << endl;
37 }
38 //函数对象可以作为参数传递
39 void doPrint(MyPrint& mp, string test)
40 {
41     mp(test);
42 }
43 void test03()
44 {
45     MyPrint myprint;
46     doPrint(myprint, "hello world");
47 }
48 int main()
49 {
50     test01();
51     test02();
52     test03();
53     system("pause");
54 }

```

## 谓词

- 返回bool类型的仿函数称为**谓词**；如果operator () 接受一个参数，那么叫做**一元谓词**；如果operator () 接受两个参数，那么叫做**二元谓词**
- 一元谓词

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 using namespace std;
5 //谓词
6 class GreaterTwo
7 {
8 public:
9     bool operator()(int val)//一元谓词
10    {
11        return val > 2;
12    }
13 };
14 void test01()
15 {
16     vector<int> v;
17     for (int i = 0; i < 5; i++)
18     {
19         v.push_back(i + 1);
20     }
21     //查找容器中是否有大于2的数字
22     //GreaterTwo为匿名的函数对象
23     vector<int>::iterator it = find_if(v.begin(), v.end(),
GreaterTwo());
24     if (it == v.end())
25     {
26         cout << "未找到" << endl;
27     }

```

```

28         else
29         {
30             cout << *it << endl;
31         }
32     }
33 }
34 int main()
35 {
36     test01();
37 }

```

- 二元谓词

```

○ 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 using namespace std;
5 //二元谓词
6 class MyCompare
7 {
8 public:
9     bool operator()(int val1, int val2)
10    {
11        return val1 > val2;
12    }
13 };
14 void test01()
15 {
16     vector<int> v1;
17     for (int i = 0; i < 10; i++)
18     {
19         v1.push_back(i + 1);
20     }
21     sort(v1.begin(), v1.end(), MyCompare());
22     for (vector<int>::iterator it = v1.begin(); it != v1.end();
23         it++)
24     {
25         cout << *it << endl;
26     }
27 }
28 int main()
29 {
30     test01();
31 }

```

## 内建函数对象

- STL中内建了部分函数对象；分为算术仿函数；关系仿函数；逻辑仿函数
- 这些仿函数所产生的对象用法和一般函数完全相同
- 使用内建的仿函数，需要引入头文件functional
- 算术仿函数
  - 功能：实现四则运算；其中negate是一元运算，其他都是二元运算

```

○ 1 #include<iostream>
2 #include<vector>

```

```

3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //算术仿函数
7  //negate一元仿函数 取反仿函数
8  void test01()
9  {
10     negate<int> n; //取反运算
11     cout << n(50) << endl;
12 }
13 //plus 二元仿函数 加法
14 void test02()
15 {
16     plus<int> p;
17     cout << p(10, 20) << endl;
18 }
19 int main()
20 {
21     test01();
22     test02();
23 }

```

- 关系仿函数：最常用的是greater

```

o 1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //关系仿函数
7  //greater 大于
8  void test01()
9  {
10     vector<int> v;
11     v.push_back(4);
12     v.push_back(8);
13     v.push_back(2);
14     v.push_back(6);
15     v.push_back(9);
16     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
17     {
18         cout << *it << " ";
19     }
20     cout << endl;
21     sort(v.begin(), v.end(), greater<int>());
22     cout << "-----" << endl;
23     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
24     {
25         cout << *it << " ";
26     }
27 }
28
29 int main()
30 {
31     test01();
32 }

```

- 逻辑仿函数（基本用不上）

```
o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //逻辑仿函数
7 //逻辑非 logical_not
8 void test01()
9 {
10     vector<bool> v;
11     v.push_back(true);
12     v.push_back(false);
13     v.push_back(true);
14     v.push_back(false);
15     v.push_back(true);
16     for (vector<bool>::iterator it = v.begin(); it != v.end();
it++)
17     {
18         cout << *it << " ";
19     }
20     cout << endl;
21     //利用逻辑非将容器搬运到容器v2中，并执行取反操作
22     vector<bool> v2;
23     v2.resize(v.size());
24     transform(v.begin(), v.end(), v2.begin(), logical_not<bool>());
25     for (vector<bool>::iterator it = v2.begin(); it != v2.end();
it++)
26     {
27         cout << *it << " ";
28     }
29 }
30 int main()
31 {
32     test01();
33 }
```

## STL常用算法

- 主要是由头文件组成
- for\_each
  - 用于遍历容器

```
o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //for_each函数，遍历容器
7 void print01(int val)
8 {
9     cout << val << " ";
10 }
11 //仿函数
12 class print02
```

```

13 {
14 public:
15     void operator()(int val)
16     {
17         cout << val << " ";
18     }
19 };
20 void test01()
21 {
22     vector<int> v;
23     v.push_back(10);
24     v.push_back(20);
25     v.push_back(54);
26     v.push_back(23);
27     v.push_back(53);
28     for_each(v.begin(), v.end(), print01);
29     cout << endl;
30     for_each(v.begin(), v.end(), print02());
31 }
32 }
33 int main()
34 {
35     test01();
36 }

```

- transform: 搬运到另一个容器中

```

○ 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //transform
7 class MyPrint
8 {
9 public:
10     void operator()(int val)
11     {
12         cout << val << " ";
13     }
14 };
15 //仿函数
16 class Transform
17 {
18 public:
19     int operator()(int v)
20     {
21         return v;
22     }
23 };
24 void test01()
25 {
26     vector<int> v;
27     v.push_back(10);
28     v.push_back(20);
29     v.push_back(54);
30     v.push_back(23);

```

```

31     v.push_back(53);
32     vector<int>vTarget;
33     vTarget.resize(v.size());//目标容器需要提前开辟空间
34     transform(v.begin(), v.end(), vTarget.begin(),Transform());
35     for_each(vTarget.begin(), vTarget.end(), MyPrint());
36
37 }
38 int main()
39 {
40     test01();
41 }

```

- 搬运的目标容器必须提前开辟空间，否则无法搬运

- 常用的查找算法

- find：查找指定元素，找到返回指定元素的迭代器，找不到返回结束迭代器

```

1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional>//内建函数对象头文件
5  using namespace std;
6  //查找内置数据类型
7  void test01()
8  {
9      vector<int>v;
10     for (int i = 0; i < 10; i++)
11     {
12         v.push_back(i + 1);
13     }
14     vector<int>::iterator it = find(v.begin(), v.end(), 6);
15     if (it == v.end())
16     {
17         cout << "没有找到";
18     }
19     else
20     {
21         cout << "找到" << *it << endl;
22     }
23 }
24 //查找自定义数据类型
25 class Person
26 {
27 public:
28     Person(string name, int age)
29     {
30         this->m_name = name;
31         this->m_age = age;
32     }
33     //重载==,使底层的find知道如何对比person数据类型
34     bool operator==(const Person& p)
35     {
36         if (this->m_name == p.m_name && this->m_age == p.m_age)
37         {
38             return true;
39         }
40         else

```



```

41     {
42         return false;
43     }
44 }
45 string m_name;
46 int m_age;
47 };
48
49 void test02()
50 {
51     vector<Person> v;
52     Person p1("A", 10);
53     Person p2("B", 20);
54     Person p3("C", 40);
55     Person p4("D", 50);
56
57     v.push_back(p1);
58     v.push_back(p2);
59     v.push_back(p3);
60     v.push_back(p4);
61     vector<Person>::iterator it = find(v.begin(), v.end(), p2);
62     if (it == v.end())
63     {
64         cout << "没有找到" << endl;
65     }
66     else
67     {
68         cout << "找到这个人了" << endl;
69         cout << "姓名" << (*it).m_name << "\t年龄" << (*it).m_age <<
endl;
70     }
71 }
72 int main()
73 {
74     test01();
75     test02();
76 }

```

- find\_if-按条件查找数据

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //查找内置数据类型
7 class GreaterFive
8 {
9 public:
10     bool operator()(int val)
11     {
12         return val > 5;
13     }
14 };
15 void test01()
16 {
17     vector<int>v;

```

```

18     for (int i = 0; i < 10; i++)
19     {
20         v.push_back(i + 1);
21     }
22     vector<int>::iterator it = find_if(v.begin(), v.end(),
GreaterFive());
23     if (it == v.end())
24     {
25         cout << "不存在" << endl;
26     }
27     else
28     {
29         cout << *it << endl;
30     }
31 }
32 //查找自定义数据类型
33 class Person
34 {
35 public:
36     Person(string name, int age)
37     {
38         this->m_name = name;
39         this->m_age = age;
40     }
41     string m_name;
42     int m_age;
43 };
44 class Greater20
45 {
46 public:
47     bool operator()(Person& p)
48     {
49         return p.m_age > 20;
50     }
51 };
52 void test02()
53 {
54     vector<Person> v;
55     Person p1("A", 10);
56     Person p2("B", 20);
57     Person p3("C", 40);
58     Person p4("D", 50);
59
60     v.push_back(p1);
61     v.push_back(p2);
62     v.push_back(p3);
63     v.push_back(p4);
64     for (vector<Person>::iterator it = find_if(v.begin(), v.end(),
Greater20()); it != v.end(); it++)
65     {
66         if (it == v.end())
67         {
68             cout << "没有找到" << endl;
69         }
70         else
71         {
72             cout << "找到这个人了" << endl;
73         }
74     }
75 }

```

```

74         cout << "姓名" << (*it).m_name << "\t年龄" << (*it).m_age
    << endl;
75     }
76 }
77
78 }
79 int main()
80 {
81     test01();
82     test02();
83 }

```

- adjacent\_find 查找相邻重复元素算法

- ```

1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //查找 adjacent_find
7  void test01()
8  {
9      vector<int>v;
10     for (int i = 0; i < 10; i++)
11     {
12         v.push_back(i + 1);
13     }
14     vector<int>::iterator pos = adjacent_find(v.begin(), v.end());
15     if (pos == v.end())
16     {
17         cout << "未找到相邻重复元素" << endl;
18     }
19     else
20     {
21         cout << *pos << endl;
22     }
23 }
24 int main()
25 {
26     test01();
27 }

```

- binary\_search:查找元素是否存在

- 二分查找算法，该算法在**无序序列中不可使用**；该算法效率高

- ```

1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //查找 adjacent_find
7  void test01()
8  {
9      vector<int>v;
10     for (int i = 0; i < 10; i++)
11     {
12         v.push_back(i + 1);

```

```

13     }
14     bool ret = binary_search(v.begin(), v.end(), 4);
15     if (ret)
16     {
17         cout << "找到元素" << endl;
18     }
19     else
20     {
21         cout << "未找到元素" << endl;
22     }
23 }
24
25 int main()
26 {
27     test01();
28 }

```

- count: 统计元素个数

```

o 1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //count
7  //1.统计内置数据类型
8  void test01()
9  {
10     vector<int> v;
11     v.push_back(10);
12     v.push_back(10);
13     v.push_back(30);
14     v.push_back(40);
15     v.push_back(10);
16     int num = count(v.begin(), v.end(), 10);
17     cout << num << endl;
18 }
19 //2.统计自定义数据类型
20 class Person
21 {
22 public:
23     Person(string name, int age)
24     {
25         this->m_name = name;
26         this->m_age = age;
27     }
28     bool operator==(const Person& p)
29     {
30         if (this->m_age == p.m_age)
31         {
32             return true;
33         }
34         else
35         {
36             return false;
37         }
38     }

```

```

39     string m_name;
40     int m_age;
41 };
42 void test02()
43 {
44     vector<Person> p;
45     Person p1("赵云", 20);
46     Person p2("左慈", 30);
47     Person p3("孙权", 40);
48     Person p4("周瑜", 20);
49     Person p5("张飞", 20);
50     p.push_back(p1);
51     p.push_back(p2);
52     p.push_back(p3);
53     p.push_back(p4);
54     p.push_back(p5);
55     Person p6("诸葛亮", 20);
56     int num = count(p.begin(), p.end(), p6); //找与诸葛亮同岁数的个数
57     cout << num << endl;
58
59
60 }
61
62 int main()
63 {
64     test01();
65     test02();
66 }

```

- count\_if:按条件统计元素个数

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //count
7 //1.统计内置数据类型
8 class Greater20
9 {
10 public:
11     bool operator()(int val)
12     {
13         return val > 20;
14     }
15
16 private:
17
18 };
19
20 void test01()
21 {
22     vector<int> v;
23     v.push_back(10);
24     v.push_back(10);
25     v.push_back(30);
26     v.push_back(40);

```

```

27     v.push_back(10);
28     int num = count_if(v.begin(), v.end(), Greater20());
29     cout << "大于20的元素个数为" << num << endl;
30 }
31 //2.统计自定义数据类型
32 class Person
33 {
34 public:
35     Person(string name, int age)
36     {
37         this->m_name = name;
38         this->m_age = age;
39     }
40     string m_name;
41     int m_age;
42 };
43 class AgeGreater20
44 {
45 public:
46     bool operator()(const Person& p)
47     {
48         return p.m_age > 20;
49     }
50 };
51 void test02()
52 {
53     vector<Person> p;
54     Person p1("赵云", 20);
55     Person p2("左慈", 30);
56     Person p3("孙权", 40);
57     Person p4("周瑜", 20);
58     Person p5("张飞", 20);
59     p.push_back(p1);
60     p.push_back(p2);
61     p.push_back(p3);
62     p.push_back(p4);
63     p.push_back(p5);
64     int num = count_if(p.begin(), p.end(), AgeGreater20()); //找年龄大
        于20的个数
65     cout << num << endl;
66 }
67
68 int main()
69 {
70     test01();
71     test02();
72 }

```

- sort:对容器内元素进行排序;需要熟练掌握

```

○ 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //sort排序算法
7 void myPrint(int val)

```

```

8  {
9      cout << val << " ";
10 }
11 void test01()
12 {
13     vector<int> v;
14     v.push_back(10);
15     v.push_back(10);
16     v.push_back(30);
17     v.push_back(40);
18     v.push_back(10);
19     sort(v.begin(), v.end());
20     for_each(v.begin(), v.end(), myPrint);
21     cout << endl;
22     //由升序改为降序
23     sort(v.begin(), v.end(), greater<int>());
24     for_each(v.begin(), v.end(), myPrint);
25     cout << endl;
26 }
27 }
28 int main()
29 {
30     test01();
31 }

```

- random\_shuffle:洗牌算法, 将元素顺序打乱

```

o 1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //sort排序算法
7  void myPrint(int val)
8  {
9      cout << val << " ";
10 }
11 void test01()
12 {
13     vector<int> v;
14     for (int i = 0; i < 10; i++)
15     {
16         v.push_back(i + 1);
17     }
18     //利用洗牌算法打乱顺序
19     random_shuffle(v.begin(), v.end());
20     for_each(v.begin(), v.end(), myPrint);
21 }
22 int main()
23 {
24     srand((unsigned int)time(NULL));
25     test01();
26 }

```

- merge:将两个容器合并, 储存到另一个容器中

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //merge排序算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 void test01()
12 {
13     vector<int> v1;
14     vector<int> v2;
15     for (int i = 0; i < 10; i++)
16     {
17         v1.push_back(i);
18         v2.push_back(i + 5);
19     }
20     vector<int> vTarget;
21     vTarget.resize(v1.size() + v2.size());
22     merge(v1.begin(), v1.end(), v2.begin(), v2.end(),
23           vTarget.begin());
24     for_each(vTarget.begin(), vTarget.end(), myPrint);
25 }
26 int main()
27 {
28     test01();
29 }

```

- reverse: 反转

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //merge排序算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 void test01()
12 {
13     vector<int> v1;
14     vector<int> v2;
15     for (int i = 0; i < 10; i++)
16     {
17         v1.push_back(i);
18     }
19     cout << "反转前" << endl;
20     for_each(v1.begin(), v1.end(), myPrint);
21     cout << endl;
22     cout << "反转后" << endl;
23     reverse(v1.begin(), v1.end());
24     for_each(v1.begin(), v1.end(), myPrint);
25 }

```



```

26 }
27 int main()
28 {
29     test01();
30 }

```

- copy:将容器内的指定范围的元素拷贝到另一容器中

```

○ 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //merge排序算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 void test01()
12 {
13     vector<int> v1;
14     vector<int> v2;
15     for (int i = 0; i < 10; i++)
16     {
17         v1.push_back(i);
18     }
19     v2.resize(v1.size());
20     copy(v1.begin(), v1.end(), v2.begin());
21     for_each(v2.begin(), v2.end(), myPrint);
22 }
23 int main()
24 {
25     test01();
26 }

```

- replace:将容器里面的旧元素替换为新元素

```

○ 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<functional> //内建函数对象头文件
5 using namespace std;
6 //merge排序算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 void test01()
12 {
13     vector<int> v1;
14     vector<int> v2;
15     for (int i = 0; i < 10; i++)
16     {
17         v1.push_back(i);
18     }
19     cout << "替换前" << endl;
20     for_each(v1.begin(), v1.end(), myPrint);

```

```

21     replace(v1.begin(), v1.end(), 4, 20);
22     cout << endl;
23     cout << "替换后" << endl;
24     for_each(v1.begin(), v1.end(), myPrint);
25 }
26 int main()
27 {
28     test01();
29 }

```

- replace\_if:将满足条件的元素替换为新元素

```

o 1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //merge排序算法
7  void myPrint(int val)
8  {
9      cout << val << " ";
10 }
11 class Greater
12 {
13 public:
14     bool operator()(int val)
15     {
16         return val > 5;
17     }
18 };
19 void test01()
20 {
21     vector<int> v1;
22     vector<int> v2;
23     for (int i = 0; i < 10; i++)
24     {
25         v1.push_back(i);
26     }
27     cout << "替换前" << endl;
28     for_each(v1.begin(), v1.end(), myPrint);
29     replace_if(v1.begin(), v1.end(), Greater(), 100);
30     cout << "替换后" << endl;
31     for_each(v1.begin(), v1.end(), myPrint);
32 }
33 int main()
34 {
35     test01();
36 }

```

- swap:互换

```

o 1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  #include<functional> //内建函数对象头文件
5  using namespace std;
6  //merge排序算法

```

```

7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 class Greater
12 {
13 public:
14     bool operator()(int val)
15     {
16         return val > 5;
17     }
18 };
19 void test01()
20 {
21     vector<int> v1;
22     vector<int> v2;
23     for (int i = 0; i < 10; i++)
24     {
25         v1.push_back(i);
26         v2.push_back(i + 32);
27     }
28     cout << "交换前" << endl;
29     for_each(v1.begin(), v1.end(), myPrint);
30     cout << endl;
31     for_each(v2.begin(), v2.end(), myPrint);
32     cout << endl;
33     cout << "交换后" << endl;
34     swap(v1, v2);
35     for_each(v1.begin(), v1.end(), myPrint);
36     cout << endl;
37     for_each(v2.begin(), v2.end(), myPrint);
38 }
39 int main()
40 {
41     test01();
42 }

```

- accumulate:将容器内元素累加; 在头文件numeric中

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<numeric>
5 using namespace std;
6 //accumulate
7 void test01()
8 {
9     vector<int> v1;
10     for (int i = 0; i <= 100; i++)
11     {
12         v1.push_back(i);
13     }
14     int sum = accumulate(v1.begin(), v1.end(), 0); //最后一个参数是起始
15     值
16     cout << sum << endl;
17 }

```

```

18
19 int main()
20 {
21     test01();
22 }

```

- fill:向容器中填充元素

```

○ 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<numeric>
5 using namespace std;
6 //fill算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 class Greater
12 {
13 public:
14     bool operator()(int val)
15     {
16         return val > 5;
17     }
18 };
19 void test01()
20 {
21     vector<int> v1;
22     v1.resize(10);
23     fill(v1.begin(), v1.end(), 100);
24     for_each(v1.begin(), v1.end(), myPrint);
25 }
26
27 int main()
28 {
29     test01();
30 }

```

- set\_intersection:求两个容器的交集

```

○ 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<numeric>
5 using namespace std;
6 //常用集合算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 class Greater
12 {
13 public:
14     bool operator()(int val)
15     {
16         return val > 5;

```

```

17     }
18 };
19 void test01()
20 {
21     vector<int> v1;
22     vector<int> v2;
23     for (int i = 0; i < 5; i++)
24     {
25         v1.push_back(i);
26         v2.push_back(i + 2);
27     }
28     //目标容器
29     vector<int> vTarget;
30     //目标容器需要提前开辟空间
31     //最特殊的情况：开辟空间取小值
32     vTarget.resize(min(v1.size(), v2.size()));
33     //获取交集
34     vector<int>::iterator itEnd = set_intersection(v1.begin(),
35     v1.end(), v2.begin(), v2.end(), vTarget.begin());
36     for_each(vTarget.begin(), itEnd, myPrint);// !!!!
37 }
38 int main()
39 {
40     test01();
41 }

```

- set\_union:求两个集合的并集

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<numeric>
5 using namespace std;
6 //常用集合算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 class Greater
12 {
13 public:
14     bool operator()(int val)
15     {
16         return val > 5;
17     }
18 };
19 void test01()
20 {
21     vector<int> v1;
22     vector<int> v2;
23     for (int i = 0; i < 5; i++)
24     {
25         v1.push_back(i);
26         v2.push_back(i + 2);
27     }
28     //目标容器

```

```

29     vector<int> vTarget;
30     //目标容器需要提前开辟空间
31     //最特殊的情况：开辟空间取小值
32     vTarget.resize(v1.size()+v2.size());
33     //获取并集
34     vector<int>::iterator itEnd = set_union(v1.begin(), v1.end(),
v2.begin(), v2.end(), vTarget.begin());
35     for_each(vTarget.begin(), itEnd, myPrint);// !!!!
36 }
37 int main()
38 {
39     test01();
40 }

```

- set\_different:求两个集合的差集

```

o 1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<numeric>
5 using namespace std;
6 //常用集合算法
7 void myPrint(int val)
8 {
9     cout << val << " ";
10 }
11 class Greater
12 {
13 public:
14     bool operator()(int val)
15     {
16         return val > 5;
17     }
18 };
19 void test01()
20 {
21     vector<int> v1;
22     vector<int> v2;
23     for (int i = 0; i < 5; i++)
24     {
25         v1.push_back(i);
26         v2.push_back(i + 2);
27     }
28     //目标容器
29     vector<int> vTarget;
30     //目标容器需要提前开辟空间
31     //最特殊的情况：开辟空间取小值
32     vTarget.resize(max(v1.size(),v2.size()));
33     //获取差集
34     vector<int>::iterator itEnd = set_difference(v1.begin(),
v1.end(), v2.begin(), v2.end(), vTarget.begin());
35     for_each(vTarget.begin(), itEnd, myPrint);// !!!!
36 }
37 int main()
38 {
39     test01();
40 }

```

