

C++课程安排

- 明确C++课程学习阶段以及课程内容

阶段	内容	目标	案例
第一阶段	C++基础语法入门	对C++有初步了解，能够有基础编程能力	通讯录管理系统
第二阶段	C++核心编程	介绍C++面向对象编程，为大型项目做铺垫	职工管理系统
第三阶段	C++提高编程	介绍C++泛型编程思想，以及STL的基本使用	演讲比赛系统

- 综合大案例：机房预约系统

C++基础入门

1 C++初识

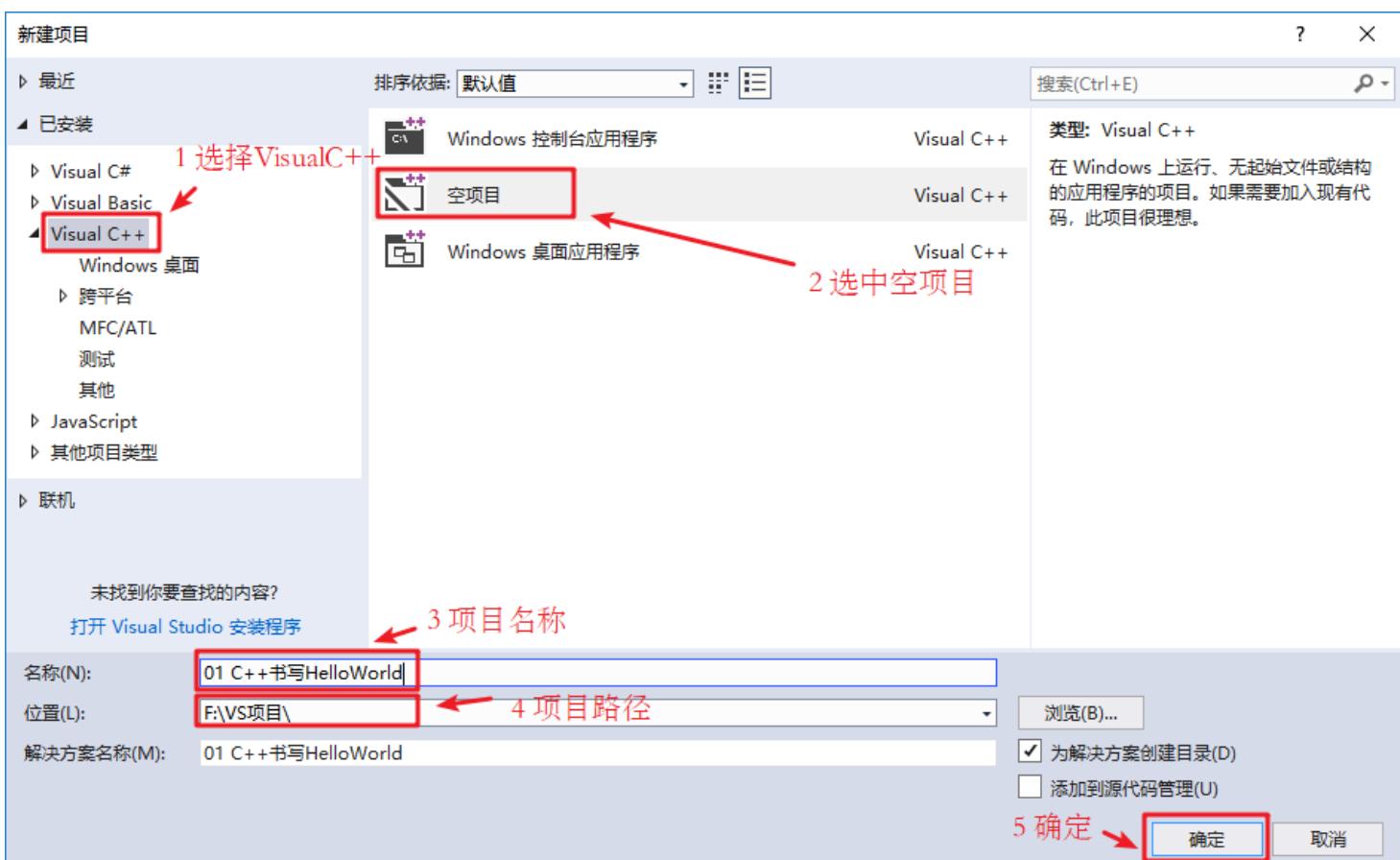
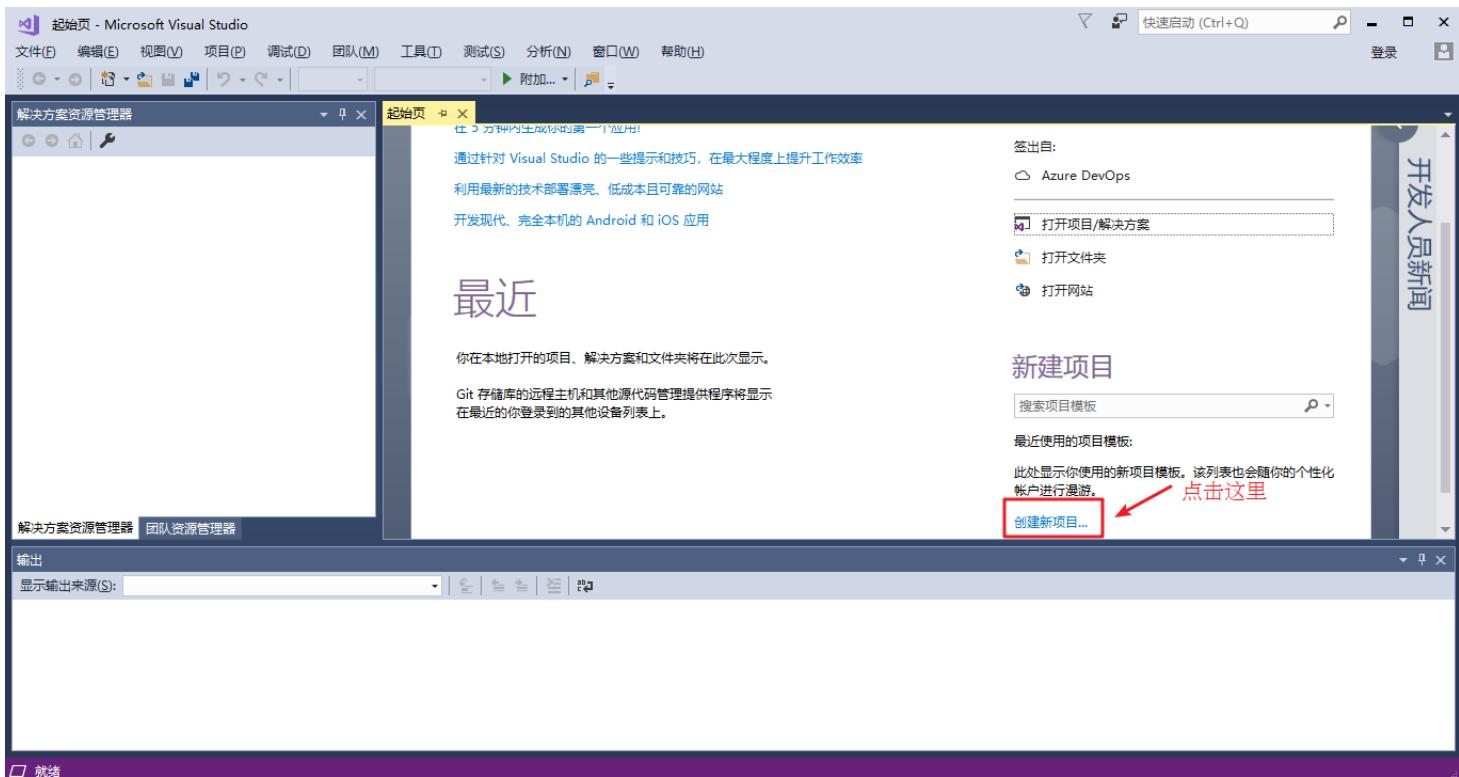
1.1 第一个C++程序

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

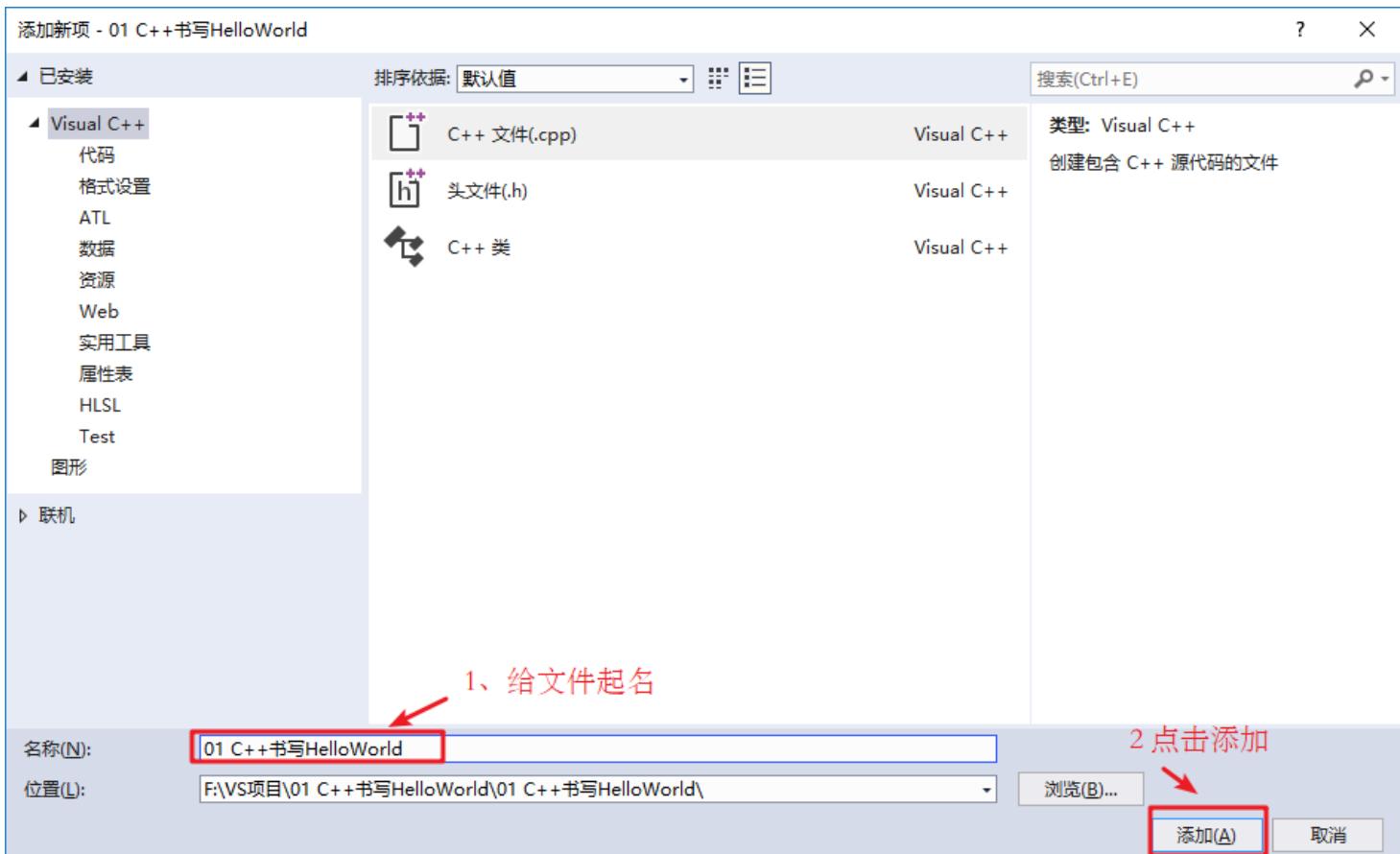


1.1.2 创建文件

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。

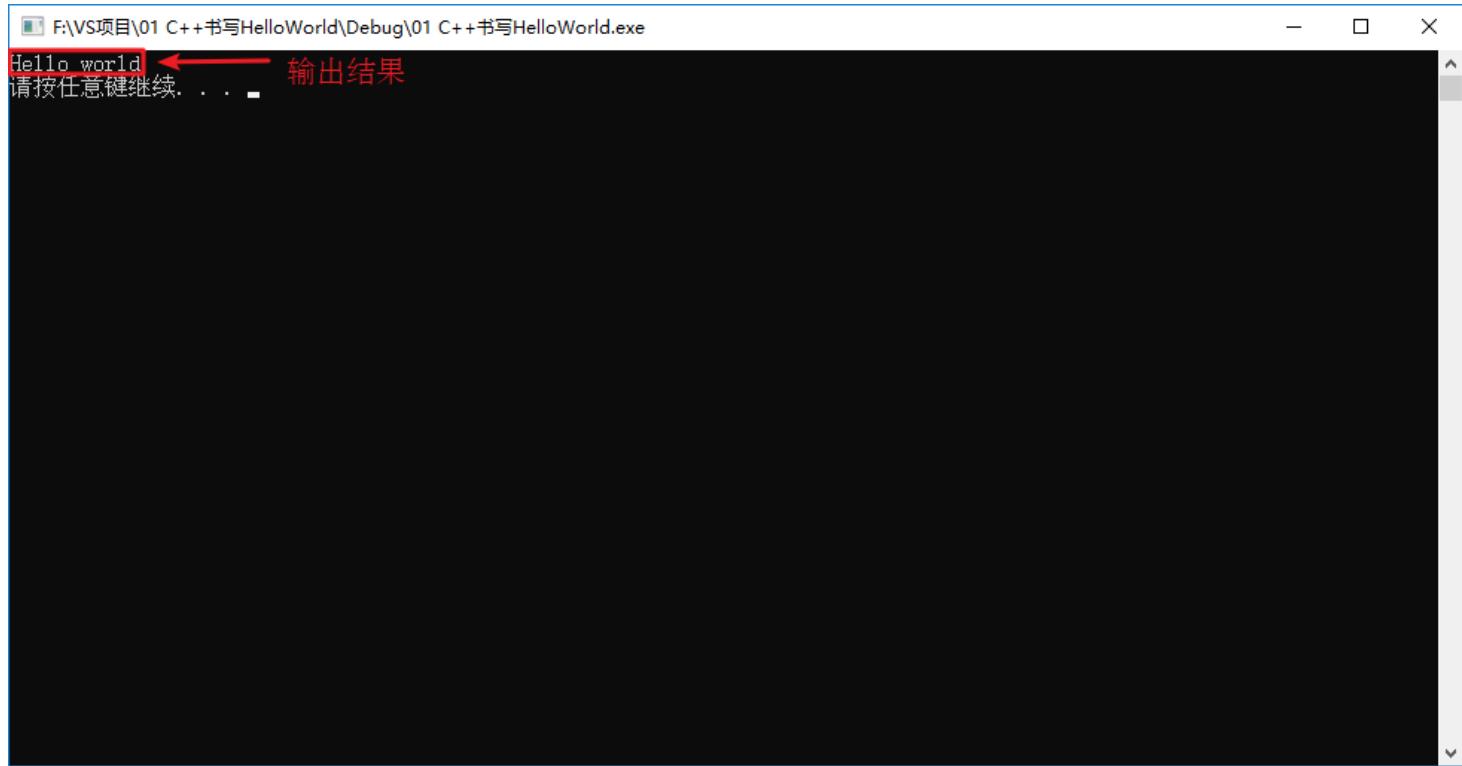


1.1.3 编写代码

```
#include<iostream>
using namespace std;

int main() {
    cout << "Hello world" << endl;
    system("pause");
    return 0;
}
```

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

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：

```
#include<iostream>
using namespace std;

int main() {
    //变量的定义
    //语法：数据类型 变量名 = 初始值

    int a = 10;

    cout << "a = " << a << endl;

    system("pause");

    return 0;
}
```

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、宏常量
#define day 7

int main() {

    cout << "一周里总共有 " << day << " 天" << endl;
    //day = 8; //报错，宏常量不可以修改

    //2、const修饰变量
    const int month = 12;
    cout << "一年里总共有 " << month << " 个月份" << endl;
    //month = 24; //报错，常量是不可以修改的

    system("pause");

    return 0;
}

```

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	

asm	do	if	return	typedef
-----	----	----	--------	---------

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。

static --修饰局部变量，局部变量的生命周期延长了，放到局部变量前，每次调动这个函数，都不会重新创建，比如一个函数内部 static int a = 2; a++; 第一次调用a =2；第二次调用a =3，出了作用域a 不会销毁，后续调用会延续之前的a的数值，static int a = 2只会调用一次，下次调用直接跳过这行。

static--修饰全局变量，全局变量会被限制，使全局变量作用域变小，静态的全局变量只能在自己所在的源文件的内部使用，出了源文件用extern函数也无法调用该全局变量。

static--修饰函数，效果和修饰全局变量相似，static修饰函数改变了函数的链接属性。

函数的定义和声明默认情况下是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。好处：

- <1> 其他文件中可以定义相同名字的函数，不会发生冲突
- <2> 静态函数不能被其他文件所用。

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 ~ 2^15-1)
int(整型)	4字节	(-2^31 ~ 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 ~ 2^31-1)

数据类型	占用空间	取值范围
long long(长长整形)	8字节	(-2^63 ~ 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：

```
int main() {
    cout << "short 类型所占内存空间为: " << sizeof(short) << endl;
    cout << "int 类型所占内存空间为: " << sizeof(int) << endl;
    cout << "long 类型所占内存空间为: " << sizeof(long) << endl;
    cout << "long long 类型所占内存空间为: " << sizeof(long long) << endl;
    system("pause");
    return 0;
}
```

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：

```
int main() {  
  
    float f1 = 3.14f;  
    double d1 = 3.14;  
  
    cout << f1 << endl;  
    cout << d1 << endl;  
  
    cout << "float sizeof = " << sizeof(f1) << endl;  
    cout << "double sizeof = " << sizeof(d1) << endl;  
  
    //科学计数法  
    float f2 = 3e2; // 3 * 10 ^ 2  
    cout << "f2 = " << f2 << endl;  
  
    float f3 = 3e-2; // 3 * 0.1 ^ 2  
    cout << "f3 = " << f3 << endl;  
  
    system("pause");  
  
    return 0;  
}
```

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：

```

int main() {

    char ch = 'a';
    cout << ch << endl;
    cout << sizeof(char) << endl;

    //ch = "abcde"; //错误，不可以用双引号
    //ch = 'abcde'; //错误，单引号内只能引用一个字符

    cout << (int)ch << endl; //查看字符a对应的ASCII码
    ch = 97; //可以直接用ASCII给字符型变量赋值
    cout << ch << endl;

    system("pause");

    return 0;
}

```

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

ASCII值	控制字符	ASCII值	字符	ASCII值	字符	ASCII值	字符
16	DLE	48	0	80	P	112	p
17	DCI	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	}
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）
\a	警报	007
\b	退格(BS)，将当前位置移到前一列	008

转义字符	含义	ASCII码值（十进制）
\f	换页(FF), 将当前位置移到下页开头	012
\n	换行(LF) , 将当前位置移到下一行开头	010
\r	回车(CR) , 将当前位置移到本行开头	013
\t	水平制表(HT) (跳到下一个TAB位置)	009
\v	垂直制表(VT)	011
\	代表一个反斜线字符”\”	092
'	代表一个单引号（撇号）字符	039
"	代表一个双引号字符	034
?	代表一个问号	063
\0	数字0	000
\ddd	8进制转义字符，d范围0~7	3位8进制
\xhh	16进制转义字符，h范围0_9, af, A~F	3位16进制

示例：

```
int main() {
    cout << "\\\" << endl;
    cout << "\\tHello" << endl;
    cout << "\\n" << endl;

    system("pause");
    return 0;
}
```

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：

```
int main() {  
    char str1[] = "hello world";  
    cout << str1 << endl;  
  
    system("pause");  
  
    return 0;  
}
```

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：

```
int main() {  
  
    string str = "hello world";  
    cout << str << endl;  
  
    system("pause");  
  
    return 0;  
}
```

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：

```
int main() {  
  
    bool flag = true;  
    cout << flag << endl; // 1  
  
    flag = false;  
    cout << flag << endl; // 0  
  
    cout << "size of bool = " << sizeof(bool) << endl; //1  
  
    system("pause");  
  
    return 0;  
}
```

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：

```

int main(){
    //整型输入
    int a = 0;
    cout << "请输入整型变量: " << endl;
    cin >> a;
    cout << a << endl;

    //浮点型输入
    double d = 0;
    cout << "请输入浮点型变量: " << endl;
    cin >> d;
    cout << d << endl;

    //字符型输入
    char ch = 0;
    cout << "请输入字符型变量: " << endl;
    cin >> ch;
    cout << ch << endl;

    //字符串型输入
    string str;
    cout << "请输入字符串型变量: " << endl;
    cin >> str;
    cout << str << endl;

    //布尔类型输入
    bool flag = true;
    cout << "请输入布尔型变量: " << endl;
    cin >> flag;
    cout << flag << endl;
    system("pause");
    return EXIT_SUCCESS;
}

```

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：

```
//加减乘除
int main() {
    int a1 = 10;
    int b1 = 3;

    cout << a1 + b1 << endl;
    cout << a1 - b1 << endl;
    cout << a1 * b1 << endl;
    cout << a1 / b1 << endl; //两个整数相除结果依然是整数

    int a2 = 10;
    int b2 = 20;
    cout << a2 / b2 << endl;

    int a3 = 10;
    int b3 = 0;
    //cout << a3 / b3 << endl; //报错，除数不可以为0

    //两个小数可以相除
    double d1 = 0.5;
    double d2 = 0.25;
    cout << d1 / d2 << endl;

    system("pause");
}

return 0;
}
```

| 总结：在除法运算中，除数不能为0

示例2：

```
//取模
int main() {
    int a1 = 10;
    int b1 = 3;

    cout << 10 % 3 << endl;

    int a2 = 10;
    int b2 = 20;

    cout << a2 % b2 << endl;

    int a3 = 10;
    int b3 = 0;

    //cout << a3 % b3 << endl; //取模运算时，除数也不能为0

    //两个小数不可以取模
    double d1 = 3.14;
    double d2 = 1.1;

    //cout << d1 % d2 << endl;

    system("pause");
    return 0;
}
```

| 总结：只有整型变量可以进行取模运算

示例3：

```

//递增
int main() {

    //后置递增
    int a = 10;
    a++; //等价于a = a + 1
    cout << a << endl; // 11

    //前置递增
    int b = 10;
    ++b;
    cout << b << endl; // 11

    //区别
    //前置递增先对变量进行++, 再计算表达式
    int a2 = 10;
    int b2 = ++a2 * 10;
    cout << b2 << endl;

    //后置递增先计算表达式, 后对变量进行++
    int a3 = 10;
    int b3 = a3++ * 10;
    cout << b3 << endl;

    system("pause");

    return 0;
}

```

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：

```
int main() {  
  
    //赋值运算符  
  
    // =  
    int a = 10;  
    a = 100;  
    cout << "a = " << a << endl;  
  
    // +=  
    a = 10;  
    a += 2; // a = a + 2;  
    cout << "a = " << a << endl;  
  
    // -=  
    a = 10;  
    a -= 2; // a = a - 2  
    cout << "a = " << a << endl;  
  
    // *=  
    a = 10;  
    a *= 2; // a = a * 2  
    cout << "a = " << a << endl;  
  
    // /=  
    a = 10;  
    a /= 2; // a = a / 2;  
    cout << "a = " << a << endl;  
  
    // %=  
    a = 10;  
    a %= 2; // a = a % 2;  
    cout << "a = " << a << endl;  
  
    system("pause");  
  
    return 0;  
}
```

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
==	相等于	4 == 3	0
!=	不等于	4 != 3	1

运算符	术语	示例	结果
<	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：

```
int main() {
    int a = 10;
    int b = 20;

    cout << (a == b) << endl; // 0
    cout << (a != b) << endl; // 1
    cout << (a > b) << endl; // 0
    cout << (a < b) << endl; // 1
    cout << (a >= b) << endl; // 0
    cout << (a <= b) << endl; // 1
    system("pause");
    return 0;
}
```

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

```
//逻辑运算符 --- 非
int main() {

    int a = 10;

    cout << !a << endl; // 0

    cout << !!a << endl; // 1

    system("pause");

    return 0;
}
```

| 总结： 真变假， 假变真

示例2：逻辑与

```
//逻辑运算符 --- 与
int main() {

    int a = 10;
    int b = 10;

    cout << (a && b) << endl;// 1

    a = 10;
    b = 0;

    cout << (a && b) << endl;// 0

    a = 0;
    b = 0;

    cout << (a && b) << endl;// 0

    system("pause");

    return 0;
}
```

| 总结： 逻辑与运算符总结： 同真为真， 其余为假

示例3：逻辑或

```
//逻辑运算符 --- 或
int main() {

    int a = 10;
    int b = 10;

    cout << (a || b) << endl;// 1

    a = 0;
    b = 0;

    cout << (a || b) << endl;// 1

    a = 0;
    b = 0;

    cout << (a || b) << endl;// 0

    system("pause");

    return 0;
}
```

| 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 }

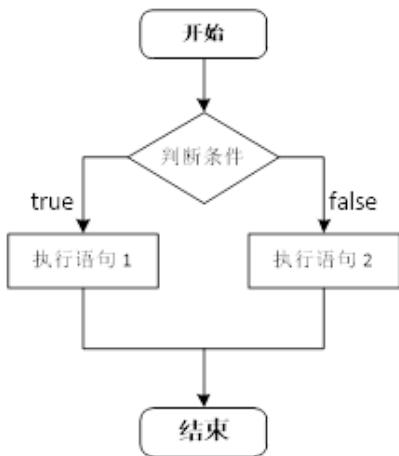


示例：

```
int main() {  
  
    //选择结构-单行if语句  
    //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  
  
    int score = 0;  
    cout << "请输入一个分数：" << endl;  
    cin >> score;  
  
    cout << "您输入的分数为：" << score << endl;  
  
    //if语句  
    //注意事项，在if判断语句后面，不要加分号  
    if (score > 600)  
    {  
        cout << "我考上了一本大学！！！" << endl;  
    }  
  
    system("pause");  
  
    return 0;  
}
```

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：

```

int main() {
    int score = 0;

    cout << "请输入考试分数：" << endl;
    cin >> score;

    if (score > 600)
    {
        cout << "我考上了一本大学" << endl;
    }
    else
    {
        cout << "我未考上一本大学" << endl;
    }

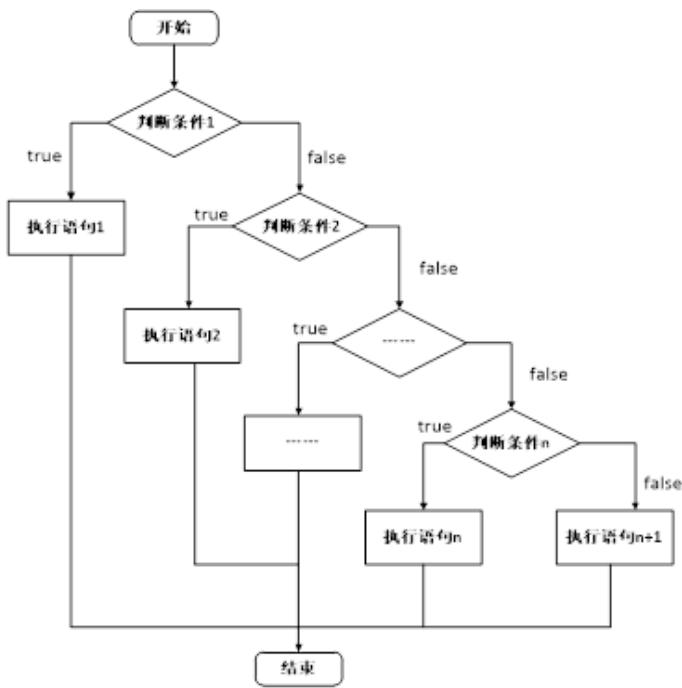
    system("pause");
}

return 0;
}

```

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：

```
int main() {
    int score = 0;
    cout << "请输入考试分数: " << endl;
    cin >> score;

    if (score > 600)
    {
        cout << "我考上了一本大学" << endl;
    }
    else if (score > 500)
    {
        cout << "我考上了二本大学" << endl;
    }
    else if (score > 400)
    {
        cout << "我考上了三本大学" << endl;
    }
    else
    {
        cout << "我未考上本科" << endl;
    }

    system("pause");
    return 0;
}
```

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：

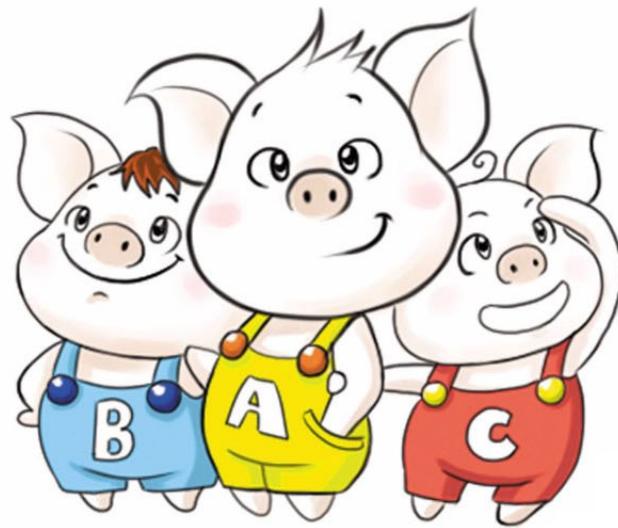
```
int main() {
    int score = 0;
    cout << "请输入考试分数: " << endl;
    cin >> score;

    if (score > 600)
    {
        cout << "我考上了一本大学" << endl;
        if (score > 700)
        {
            cout << "我考上了北大" << endl;
        }
        else if (score > 650)
        {
            cout << "我考上了清华" << endl;
        }
        else
        {
            cout << "我考上了人大" << endl;
        }
    }
    else if (score > 500)
    {
        cout << "我考上了二本大学" << endl;
    }
    else if (score > 400)
    {
        cout << "我考上了三本大学" << endl;
    }
    else
    {
        cout << "我未考上本科" << endl;
    }

    system("pause");
    return 0;
}
```

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？



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：

```
int main() {  
  
    int a = 10;  
    int b = 20;  
    int c = 0;  
  
    c = a > b ? a : b;  
    cout << "c = " << c << endl;  
  
    //C++中三目运算符返回的是变量,可以继续赋值  
  
    (a > b ? a : b) = 100;  
  
    cout << "a = " << a << endl;  
    cout << "b = " << b << endl;  
    cout << "c = " << c << endl;  
  
    system("pause");  
  
    return 0;  
}
```

| 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：

```
switch(表达式)  
{  
    case 结果1: 执行语句; break;  
    case 结果2: 执行语句; break;  
    ...  
    default: 执行语句; break;  
}
```

示例：

```

int main() {

    //请给电影评分
    //10 ~ 9 经典
    // 8 ~ 7 非常好
    // 6 ~ 5 一般
    // 5分以下 烂片

    int score = 0;
    cout << "请给电影打分" << endl;
    cin >> score;

    switch (score)
    {
        case 10:
        case 9:
            cout << "经典" << endl;
            break;
        case 8:
            cout << "非常好" << endl;
            break;
        case 7:
        case 6:
            cout << "一般" << endl;
            break;
        default:
            cout << "烂片" << endl;
            break;
    }

    system("pause");

    return 0;
}

```

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

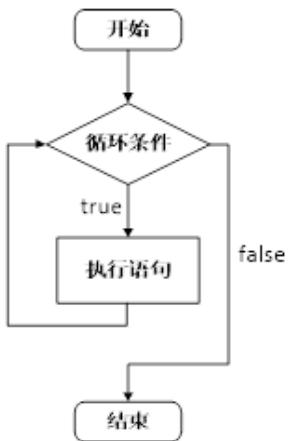
4.2 循环结构

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：

```

int main() {

    int num = 0;
    while (num < 10)
    {
        cout << "num = " << num << endl;
        num++;
    }

    system("pause");

    return 0;
}

```

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。

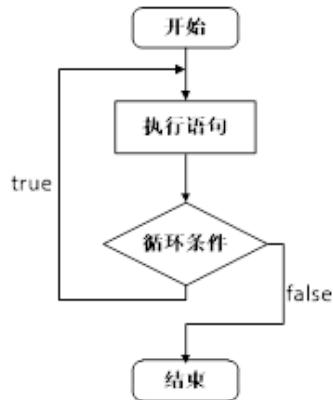


4.2.2 do...while循环语句

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：

```
int main() {  
    int num = 0;  
  
    do  
    {  
        cout << num << endl;  
        num++;  
  
    } while (num < 10);  
  
    system("pause");  
  
    return 0;  
}
```

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：

```
int main() {
    for (int i = 0; i < 10; i++)
    {
        cout << i << endl;
    }

    system("pause");
    return 0;
}
```

详解：

```
int main() {
    执行一次 → 0      1      3
    for (int i = 0; i < 10; i++)
    {
        2 cout << i << endl;
    }
}

执行顺序: 0123123123...
```

```
    system("pause");

    return 0;
}
```

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。



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：

```

int main() {
    //外层循环执行1次，内层循环执行1轮
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            cout << "*" << " ";
        }
        cout << endl;
    }

    system("pause");

    return 0;
}

```

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



乘法口诀表

$1 \times 1 = 1$									
$1 \times 2 = 2$	$2 \times 2 = 4$								
$1 \times 3 = 3$	$2 \times 3 = 6$	$3 \times 3 = 9$							
$1 \times 4 = 4$	$2 \times 4 = 8$	$3 \times 4 = 12$	$4 \times 4 = 16$						
$1 \times 5 = 5$	$2 \times 5 = 10$	$3 \times 5 = 15$	$4 \times 5 = 20$	$5 \times 5 = 25$					
$1 \times 6 = 6$	$2 \times 6 = 12$	$3 \times 6 = 18$	$4 \times 6 = 24$	$5 \times 6 = 30$	$6 \times 6 = 36$				
$1 \times 7 = 7$	$2 \times 7 = 14$	$3 \times 7 = 21$	$4 \times 7 = 28$	$5 \times 7 = 35$	$6 \times 7 = 42$	$7 \times 7 = 49$			
$1 \times 8 = 8$	$2 \times 8 = 16$	$3 \times 8 = 24$	$4 \times 8 = 32$	$5 \times 8 = 40$	$6 \times 8 = 48$	$7 \times 8 = 56$	$8 \times 8 = 64$		
$1 \times 9 = 9$	$2 \times 9 = 18$	$3 \times 9 = 27$	$4 \times 9 = 36$	$5 \times 9 = 45$	$6 \times 9 = 54$	$7 \times 9 = 63$	$8 \times 9 = 72$	$9 \times 9 = 81$	

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:

```
int main() {
    //1、在switch 语句中使用break
    cout << "请选择您挑战副本的难度: " << endl;
    cout << "1、普通" << endl;
    cout << "2、中等" << endl;
    cout << "3、困难" << endl;

    int num = 0;

    cin >> num;

    switch (num)
    {
        case 1:
            cout << "您选择的是普通难度" << endl;
            break;
        case 2:
            cout << "您选择的是中等难度" << endl;
            break;
        case 3:
            cout << "您选择的是困难难度" << endl;
            break;
    }

    system("pause");

    return 0;
}
```

示例2:

```

int main() {
    //2、在循环语句中用break
    for (int i = 0; i < 10; i++)
    {
        if (i == 5)
        {
            break; //跳出循环语句
        }
        cout << i << endl;
    }

    system("pause");

    return 0;
}

```

示例3：

```

int main() {
    //在嵌套循环语句中使用break，退出内层循环
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            if (j == 5)
            {
                break;
            }
            cout << "*" << " ";
        }
        cout << endl;
    }

    system("pause");

    return 0;
}

```

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：

```
int main() {  
  
    for (int i = 0; i < 100; i++)  
    {  
        if (i % 2 == 0)  
        {  
            continue;  
        }  
        cout << i << endl;  
    }  
  
    system("pause");  
  
    return 0;  
}
```

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：

```
int main() {  
  
    cout << "1" << endl;  
  
    goto FLAG;  
  
    cout << "2" << endl;  
    cout << "3" << endl;  
    cout << "4" << endl;  
  
FLAG:  
  
    cout << "5" << endl;  
  
    system("pause");  
  
    return 0;  
}
```

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



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

```

int main() {

    //定义方式1
    //数据类型 数组名[元素个数];
    int score[10];

    //利用下标赋值
    score[0] = 100;
    score[1] = 99;
    score[2] = 85;

    //利用下标输出
    cout << score[0] << endl;
    cout << score[1] << endl;
    cout << score[2] << endl;

    //第二种定义方式
    //数据类型 数组名[元素个数] = {值1, 值2 , 值3 ...};
    //如果{}内不足10个数据, 剩余数据用0补全
    int score2[10] = { 100, 90, 80, 70, 60, 50, 40, 30, 20, 10 };

    //逐个输出
    //cout << score2[0] << endl;
    //cout << score2[1] << endl;

    //一个一个输出太麻烦, 因此可以利用循环进行输出
    for (int i = 0; i < 10; i++)
    {
        cout << score2[i] << endl;
    }

    //定义方式3
    //数据类型 数组名[] = {值1, 值2 , 值3 ...};
    int score3[] = { 100, 90, 80, 70, 60, 50, 40, 30, 20, 10 };

    for (int i = 0; i < 10; i++)
    {
        cout << score3[i] << endl;
    }

    system("pause");

    return 0;
}

```

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：

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

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**

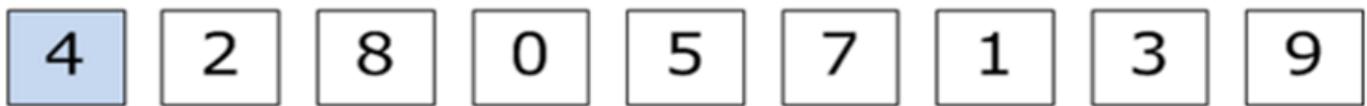
****案例描述：**请声明一个5个元素的数组，并且将元素逆置.**

(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



示例： 将数组 { 4,2,8,0,5,7,1,3,9 } 进行升序排序

```

int main() {

    int arr[9] = { 4, 2, 8, 0, 5, 7, 1, 3, 9 };

    for (int i = 0; i < 9 - 1; i++)
    {
        for (int j = 0; j < 9 - 1 - i; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    for (int i = 0; i < 9; i++)
    {
        cout << arr[i] << endl;
    }

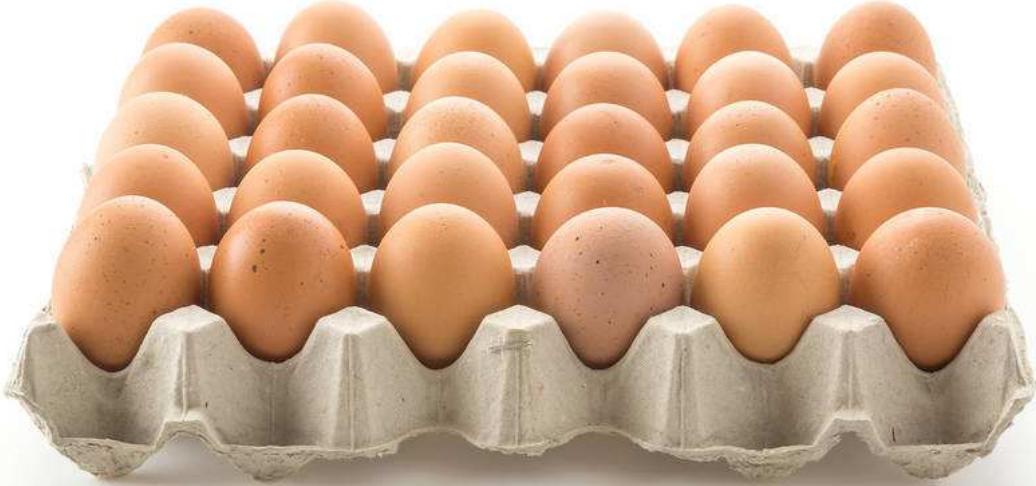
    system("pause");
}

return 0;
}

```

5.3 二维数组

二维数组就是在一维数组上，多加一个维度。



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：

```

int main() {

    //方式1
    //数组类型 数组名 [行数][列数]
    int arr[2][3];
    arr[0][0] = 1;
    arr[0][1] = 2;
    arr[0][2] = 3;
    arr[1][0] = 4;
    arr[1][1] = 5;
    arr[1][2] = 6;

    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    //方式2
    //数据类型 数组名[行数][列数] = { {数据1, 数据2} , {数据3, 数据4} };
    int arr2[2][3] =
    {
        {1, 2, 3},
        {4, 5, 6}
    };

    //方式3
    //数据类型 数组名[行数][列数] = { 数据1, 数据2 ,数据3, 数据4 } ;
    int arr3[2][3] = { 1, 2, 3, 4, 5, 6 };

    //方式4
    //数据类型 数组名[][][列数] = { 数据1, 数据2 ,数据3, 数据4 } ;
    int arr4[][3] = { 1, 2, 3, 4, 5, 6 };

    system("pause");
}

return 0;
}

```

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：

```

int main() {

    //二维数组数组名
    int arr[2][3] =
    {
        {1, 2, 3},
        {4, 5, 6}
    };

    cout << "二维数组大小: " << sizeof(arr) << endl;
    cout << "二维数组一行大小: " << sizeof(arr[0]) << endl;
    cout << "二维数组元素大小: " << sizeof(arr[0][0]) << endl;

    cout << "二维数组行数: " << sizeof(arr) / sizeof(arr[0]) << endl;
    cout << "二维数组列数: " << sizeof(arr[0]) / sizeof(arr[0][0]) << endl;

    //地址
    cout << "二维数组首地址: " << arr << endl;
    cout << "二维数组第一行地址: " << arr[0] << endl;
    cout << "二维数组第二行地址: " << arr[1] << endl;

    cout << "二维数组第一个元素地址: " << &arr[0][0] << endl;
    cout << "二维数组第二个元素地址: " << &arr[0][1] << endl;

    system("pause");

    return 0;
}

```

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：

```
int main() {
    int scores[3][3] =
    {
        {100, 100, 100},
        {90, 50, 100},
        {60, 70, 80},
    };
    string names[3] = { "张三", "李四", "王五" };

    for (int i = 0; i < 3; i++)
    {
        int sum = 0;
        for (int j = 0; j < 3; j++)
        {
            sum += scores[i][j];
        }
        cout << names[i] << "同学总成绩为: " << sum << endl;
    }

    system("pause");
    return 0;
}
```

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：

返回值类型 函数名 (参数列表)

{

函数体语句

return表达式

}

- 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

```
//函数定义
int add(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

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：

```
//函数定义
int add(int num1, int num2) //定义中的num1, num2称为形式参数，简称形参
{
    int sum = num1 + num2;
    return sum;
}

int main() {
    int a = 10;
    int b = 10;
    //调用add函数
    int sum = add(a, b); //调用时的a, b称为实际参数，简称实参
    cout << "sum = " << sum << endl;

    a = 100;
    b = 100;

    sum = add(a, b);
    cout << "sum = " << sum << endl;

    system("pause");
    return 0;
}
```

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：

```

void swap(int num1, int num2)
{
    cout << "交换前: " << endl;
    cout << "num1 = " << num1 << endl;
    cout << "num2 = " << num2 << endl;

    int temp = num1;
    num1 = num2;
    num2 = temp;

    cout << "交换后: " << endl;
    cout << "num1 = " << num1 << endl;
    cout << "num2 = " << num2 << endl;

    //return ; 当函数声明时候，不需要返回值，可以不写return
}

int main() {

    int a = 10;
    int b = 20;

    swap(a, b);

    cout << "mian中的 a = " << a << endl;
    cout << "mian中的 b = " << b << endl;

    system("pause");
    return 0;
}

```

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：

```
//函数常见样式
//1、无参无返
void test01()
{
    //void a = 10; //无类型不可以创建变量,原因无法分配内存
    cout << "this is test01" << endl;
    //test01(); 函数调用
}

//2、有参无返
void test02(int a)
{
    cout << "this is test02" << endl;
    cout << "a = " << a << endl;
}

//3、无参有返
int test03()
{
    cout << "this is test03 " << endl;
    return 10;
}

//4、有参有返
int test04(int a, int b)
{
    cout << "this is test04 " << endl;
    int sum = a + b;
    return sum;
}
```

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：

```
//声明可以多次，定义只能一次
//声明
int max(int a, int b);
int max(int a, int b);
//定义
int max(int a, int b)
{
    return a > b ? a : b;
}

int main() {

    int a = 100;
    int b = 200;

    cout << max(a, b) << endl;

    system("pause");

    return 0;
}
```

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：

```
//swap.h文件
#include<iostream>
using namespace std;

//实现两个数字交换的函数声明
void swap(int a, int b);
```

```
//swap.cpp文件
#include "swap.h"

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
}

//main函数文件
#include "swap.h"
int main() {

    int a = 100;
    int b = 200;
    swap(a, b);

    system("pause");
    return 0;
}
```

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：

```
int main() {  
    //1、指针的定义  
    int a = 10; //定义整型变量a  
  
    //指针定义语法： 数据类型 * 变量名；  
    int * p;  
  
    //指针变量赋值  
    p = &a; //指针指向变量a的地址  
    cout << &a << endl; //打印数据a的地址  
    cout << p << endl; //打印指针变量p  
  
    //2、指针的使用  
    //通过*操作指针变量指向的内存  
    cout << "*p = " << *p << endl;  
  
    system("pause");  
  
    return 0;  
}
```

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：

```
int main() {  
    int a = 10;  
  
    int * p;  
    p = &a; //指针指向数据a的地址  
  
    cout << *p << endl; /* 解引用  
    cout << sizeof(p) << endl;  
    cout << sizeof(char *) << endl;  
    cout << sizeof(float *) << endl;  
    cout << sizeof(double *) << endl;  
  
    system("pause");  
  
    return 0;  
}
```

| 总结：所有指针类型在32位操作系统下是4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

示例1：空指针

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

野指针：指针变量指向非法的内存空间

示例2：野指针

```
int main() {  
    //指针变量p指向内存地址编号为0x1100的空间  
    int * p = (int *)0x1100;  
  
    //访问野指针报错  
    cout << *p << endl;  
  
    system("pause");  
  
    return 0;  
}
```

| 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既修饰指针，又修饰常量

示例：

```
int main() {  
  
    int a = 10;  
    int b = 10;  
  
    //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  
    const int * p1 = &a;  
    p1 = &b; //正确  
    /*p1 = 100; 报错  
  
    //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  
    int * const p2 = &a;  
    //p2 = &b; //错误  
    *p2 = 100; //正确  
  
    //const既修饰指针又修饰常量  
    const int * const p3 = &a;  
    //p3 = &b; //错误  
    /*p3 = 100; //错误  
  
    system("pause");  
  
    return 0;  
}
```

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：

```
int main() {  
  
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
    int * p = arr; //指向数组的指针  
  
    cout << "第一个元素: " << arr[0] << endl;  
    cout << "指针访问第一个元素: " << *p << endl;  
  
    for (int i = 0; i < 10; i++)  
    {  
        //利用指针遍历数组  
        cout << *p << endl;  
        p++;  
    }  
  
    system("pause");  
  
    return 0;  
}
```

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：

```
//值传递
void swap1(int a ,int b)
{
    int temp = a;
    a = b;
    b = temp;
}
//地址传递
void swap2(int * p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main() {
    int a = 10;
    int b = 20;
    swap1(a, b); // 值传递不会改变实参

    swap2(&a, &b); //地址传递会改变实参

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    system("pause");
    return 0;
}
```

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：

```

//冒泡排序函数
void bubbleSort(int * arr, int len) //int * arr 也可以写为int arr[]
{
    for (int i = 0; i < len - 1; i++)
    {
        for (int j = 0; j < len - 1 - i; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

//打印数组函数
void printArray(int arr[], int len)
{
    for (int i = 0; i < len; i++)
    {
        cout << arr[i] << endl;
    }
}

int main() {

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

    bubbleSort(arr, len);

    printArray(arr, len);

    system("pause");

    return 0;
}

```

| 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：

```
//结构体定义
struct student
{
    //成员列表
    string name; //姓名
    int age; //年龄
    int score; //分数
}stu3; //结构体变量创建方式3

int main() {
    //结构体变量创建方式1
    struct student stu1; //struct 关键字可以省略

    stu1.name = "张三";
    stu1.age = 18;
    stu1.score = 100;

    cout << "姓名：" << stu1.name << " 年龄：" << stu1.age << " 分数：" << stu1.score << endl;

    //结构体变量创建方式2
    struct student stu2 = { "李四", 19, 60 };

    cout << "姓名：" << stu2.name << " 年龄：" << stu2.age << " 分数：" << stu2.score << endl;

    stu3.name = "王五";
    stu3.age = 18;
    stu3.score = 80;

    cout << "姓名：" << stu3.name << " 年龄：" << stu3.age << " 分数：" << stu3.score << endl;

    system("pause");

    return 0;
}
```

| 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：

```
//结构体定义
struct student
{
    //成员列表
    string name; //姓名
    int age; //年龄
    int score; //分数
}

int main()
{
    //结构体数组
    struct student arr[3]=
    {
        {"张三", 18, 80 },
        {"李四", 19, 60 },
        {"王五", 20, 70 }
    };

    for (int i = 0; i < 3; i++)
    {
        cout << "姓名：" << arr[i].name << " 年龄：" << arr[i].age << " 分数：" << arr[i]
    }

    system("pause");
    return 0;
}
```

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：

```
//结构体定义
struct student
{
    //成员列表
    string name; //姓名
    int age; //年龄
    int score; //分数
};

int main() {
    struct student stu = { "张三", 18, 100, };

    struct student * p = &stu;

    p->score = 80; //指针通过 -> 操作符可以访问成员

    cout << "姓名: " << p->name << " 年龄: " << p->age << " 分数: " << p->score << endl;

    system("pause");

    return 0;
}
```

| 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：

```

//学生结构体定义
struct student
{
    //成员列表
    string name; //姓名
    int age; //年龄
    int score; //分数
};

//教师结构体定义
struct teacher
{
    //成员列表
    int id; //职工编号
    string name; //教师姓名
    int age; //教师年龄
    struct student stu; //子结构体 学生
};

int main() {
    struct teacher t1;
    t1.id = 10000;
    t1.name = "老王";
    t1.age = 40;

    t1.stu.name = "张三";
    t1.stu.age = 18;
    t1.stu.score = 100;

    cout << "教师 职工编号: " << t1.id << " 姓名: " << t1.name << " 年龄: " << t1.age << endl;
    cout << "辅导学员 姓名: " << t1.stu.name << " 年龄: " << t1.stu.age << " 考试分数: " << t1.stu.score << endl;
    system("pause");
    return 0;
}

```

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：

```

//学生结构体定义
struct student
{
    //成员列表
    string name; //姓名
    int age; //年龄
    int score; //分数
};

//值传递
void printStudent(student stu )
{
    stu.age = 28;
    cout << "子函数中 姓名: " << stu.name << " 年龄: " << stu.age << " 分数: " << stu.score <
}

//地址传递
void printStudent2(student *stu)
{
    stu->age = 28;
    cout << "子函数中 姓名: " << stu->name << " 年龄: " << stu->age << " 分数: " << stu->scor
}

int main() {

    student stu = { "张三", 18, 100 };
    //值传递
    printStudent(stu);
    cout << "主函数中 姓名: " << stu.name << " 年龄: " << stu.age << " 分数: " << stu.score <

    cout << endl;

    //地址传递
    printStudent2(&stu);
    cout << "主函数中 姓名: " << stu.name << " 年龄: " << stu.age << " 分数: " << stu.score <

    system("pause");

    return 0;
}

```

| 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：

```

//学生结构体定义
struct student
{
    //成员列表
    string name; //姓名
    int age; //年龄
    int score; //分数
};

//const使用场景
void printStudent(const student *stu) //加const防止函数体中的误操作
{
    //stu->age = 100; //操作失败，因为加了const修饰
    cout << "姓名：" << stu->name << " 年龄：" << stu->age << " 分数：" << stu->score << endl;
}

int main() {
    student stu = { "张三", 18, 100 };

    printStudent(&stu);

    system("pause");
    return 0;
}

```

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：

```

struct Student
{
    string name;
    int score;
};

struct Teacher
{
    string name;
    Student sArray[5];
};

void allocateSpace(Teacher tArray[], int len)
{
    string tName = "教师";
    string sName = "学生";
    string nameSeed = "ABCDE";
    for (int i = 0; i < len; i++)
    {
        tArray[i].name = tName + nameSeed[i];

        for (int j = 0; j < 5; j++)
        {
            tArray[i].sArray[j].name = sName + nameSeed[j];
            tArray[i].sArray[j].score = rand() % 61 + 40;
        }
    }
}

void printTeachers(Teacher tArray[], int len)
{
    for (int i = 0; i < len; i++)
    {
        cout << tArray[i].name << endl;
        for (int j = 0; j < 5; j++)
        {
            cout << "\t姓名: " << tArray[i].sArray[j].name << " 分数: " << tArray[i].sArray[j].score;
        }
    }
}

int main()
{
    srand((unsigned int)time(NULL)); //随机数种子 头文件 #include <ctime>

    Teacher tArray[3]; //老师数组

    int len = sizeof(tArray) / sizeof(Teacher);

    allocateSpace(tArray, len); //创建数据

    printTeachers(tArray, len); //打印数据

    system("pause");
}

```

```
    return 0;  
}
```

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：

```
{"刘备", 23, "男"},  
 {"关羽", 22, "男"},  
 {"张飞", 20, "男"},  
 {"赵云", 21, "男"},  
 {"貂蝉", 19, "女"},
```

示例：

```

//英雄结构体
struct hero
{
    string name;
    int age;
    string sex;
};

//冒泡排序
void bubbleSort(hero arr[], int len)
{
    for (int i = 0; i < len - 1; i++)
    {
        for (int j = 0; j < len - 1 - i; j++)
        {
            if (arr[j].age > arr[j + 1].age)
            {
                hero temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

//打印数组
void printHeros(hero arr[], int len)
{
    for (int i = 0; i < len; i++)
    {
        cout << "姓名: " << arr[i].name << " 性别: " << arr[i].sex << " 年龄: " << arr[i].age;
    }
}

int main() {

    struct hero arr[5] =
    {
        {"刘备", 23, "男"},  

        {"关羽", 22, "男"},  

        {"张飞", 20, "男"},  

        {"赵云", 21, "男"},  

        {"貂蝉", 19, "女"},  

    };

    int len = sizeof(arr) / sizeof(hero); //获取数组元素个数

    bubbleSort(arr, len); //排序

    printHeros(arr, len); //打印

    system("pause");

    return 0;
}

```

C++核心编程

本阶段主要针对C++面向对象编程技术做详细讲解，探讨C++中的核心和精髓。

1 内存分区模型

C++程序在执行时，将内存大方向划分为4个区域

- 代码区：存放函数体的二进制代码，由操作系统进行管理的
- 全局区：存放全局变量和静态变量以及常量
- 栈区：由编译器自动分配释放，存放函数的参数值,局部变量，局部常量等
- 堆区：由程序员分配和释放,若程序员不释放,程序结束时由操作系统回收

内存四区意义：

不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程

1.1 程序运行前

在程序编译后，生成了exe可执行程序，**未执行该程序前**分为两个区域

代码区：

存放 CPU 执行的机器指令

代码区是**共享的**，共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可

代码区是**只读的**，使其只读的原因是防止程序意外地修改了它的指令

全局区：

全局变量和静态变量存放在**此**.

全局区还包含了常量区，字符串常量和其他常量也存放在**此**.

该区域的数据在程序结束后由操作系统**释放**.

示例：

```
//全局变量
int g_a = 10;
int g_b = 10;

//全局常量
const int c_g_a = 10;
const int c_g_b = 10;

int main() {

    //局部变量
    int a = 10;
    int b = 10;

    //打印地址
    cout << "局部变量a地址为: " << (int)&a << endl;           //在64位系统中，地址为8字节，这里用int会
    cout << "局部变量b地址为: " << (int)&b << endl;

    cout << "全局变量g_a地址为: " << (int)&g_a << endl;
    cout << "全局变量g_b地址为: " << (int)&g_b << endl;

    //静态变量
    static int s_a = 10;
    static int s_b = 10;

    cout << "静态变量s_a地址为: " << (int)&s_a << endl;
    cout << "静态变量s_b地址为: " << (int)&s_b << endl;

    cout << "字符串常量地址为: " << (int)&"hello world" << endl;
    cout << "字符串常量地址为: " << (int)&"hello world1" << endl;

    cout << "全局常量c_g_a地址为: " << (int)&c_g_a << endl;
    cout << "全局常量c_g_b地址为: " << (int)&c_g_b << endl;

    const int c_l_a = 10;
    const int c_l_b = 10;
    cout << "局部常量c_l_a地址为: " << (int)&c_l_a << endl;
    cout << "局部常量c_l_b地址为: " << (int)&c_l_b << endl;

    system("pause");

    return 0;
}
```

打印结果：

局部变量a地址为: 9697232
局部变量b地址为: 9697220
全局变量g_a地址为: 3461120
全局变量g_b地址为: 3461124
静态变量s_a地址为: 3461128
静态变量s_b地址为: 3461132
字符串常量地址为: 3451880
字符串常量地址为: 3451896
全局常量c_g_a地址为: 3452096
全局常量c_g_b地址为: 3452100
局部常量c_l_a地址为: 9697208
局部常量c_l_b地址为: 9697196
请按任意键继续... . . .

Annotations pointing to specific memory addresses:

- 指向 "局部变量a地址为: 9697232" 的箭头标注为 "局部变量存放在栈区"
- 指向 "全局变量g_a地址为: 3461120" 和 "全局变量g_b地址为: 3461124" 的箭头标注为 "全局变量和静态变量存放在全局区"
- 指向 "字符串常量地址为: 3451880" 和 "字符串常量地址为: 3451896" 的箭头标注为 "常量区"
- 指向 "全局常量c_g_a地址为: 3452096" 和 "全局常量c_g_b地址为: 3452100" 的箭头标注为 "全局常量存放在全局区"
- 指向 "局部常量c_l_a地址为: 9697208" 和 "局部常量c_l_b地址为: 9697196" 的箭头标注为 "局部常量存放在栈区"

总结：

- C++中在程序运行前分为全局区和代码区
- 代码区特点是共享和只读
- 全局区中存放全局变量、静态变量、常量
- 常量区中存放 const修饰的全局常量 和 字符串常量

1.2 程序运行后

栈区：

由编译器自动分配释放, 存放函数的参数值, 局部变量等

注意事项：不要返回局部变量的地址，栈区开辟的数据由编译器自动释放

示例：

```
int * func()
{
    int a = 10;
    return &a;
}

int main() {
    int *p = func();

    cout << *p << endl; //第一次可以打印正确的数字是因为编译器做了保留，在新版编译器里，这里不会保留
    cout << *p << endl;

    system("pause");

    return 0;
}
```

堆区：

由程序员分配释放,若程序员不释放,程序结束时由操作系统回收

在C++中主要利用new在堆区开辟内存

示例：

```
int* func()
{
    int* a = new int(10);
    return a;
}

int main() {
    int *p = func();

    cout << *p << endl;
    cout << *p << endl;

    system("pause");

    return 0;
}
```

总结：

堆区数据由程序员管理开辟和释放

堆区数据利用new关键字进行开辟内存

1.3 new操作符

C++中利用new操作符在堆区开辟数据

堆区开辟的数据，由程序员手动开辟，手动释放，释放利用操作符 delete

语法： new 数据类型

利用new创建的数据，会返回该数据对应的类型的指针

示例1： 基本语法

```
int* func()
{
    int* a = new int(10);
    return a;
}

int main() {
    int *p = func();

    cout << *p << endl;
    cout << *p << endl;

    //利用delete释放堆区数据
    delete p;

    //cout << *p << endl; //报错，释放的空间不可访问，但新版的编译器输出为0? ? ? ?

    system("pause");

    return 0;
}
```

示例2： 开辟数组

```
//堆区开辟数组
int main() {

    int* arr = new int[10];

    for (int i = 0; i < 10; i++)
    {
        arr[i] = i + 100;
    }

    for (int i = 0; i < 10; i++)
    {
        cout << arr[i] << endl;
    }
    //释放数组 delete 后加 []
    delete[] arr;

    system("pause");

    return 0;
}
```

自己操作时为什么会这样？？？

释放数组 delete 后加 []

2 引用

2.1 引用的基本使用

**作用： **给变量起别名

语法： 数据类型 &别名 = 原名

示例：

```

int main() {

    int a = 10;
    int &b = a;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    b = 100;

    cout << "a = " << a << endl; //输出a=100
    cout << "b = " << b << endl;

    system("pause");

    return 0;
}

```

2.2 引用注意事项

- 引用必须初始化
- 引用在初始化后，不可以改变（不能再变成其他变量的别名）

示例：

```

int main() {

    int a = 10;
    int b = 20;
    //int &c; //错误，引用必须初始化
    int &c = a; //一旦初始化后，就不可以更改
    c = b; //这是赋值操作，不是更改引用

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;

    system("pause");

    return 0;
}

```

2.3 引用做函数参数

作用：函数传参时，可以利用引用的技术让形参修饰实参

优点：可以简化指针修改实参

示例：

```

//1. 值传递
void mySwap01(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

//2. 地址传递
void mySwap02(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

//3. 引用传递          //引用传递用的是原始数据，比起值传递不需要再copy出一份来
void mySwap03(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {

    int a = 10;
    int b = 20;

    mySwap01(a, b);
    cout << "a:" << a << " b:" << b << endl;

    mySwap02(&a, &b);
    cout << "a:" << a << " b:" << b << endl;

    mySwap03(a, b);
    cout << "a:" << a << " b:" << b << endl;

    system("pause");

    return 0;
}

```

| 总结：通过引用参数产生的效果同按地址传递是一样的。引用的语法更清楚简单

2.4 引用做函数返回值

作用：引用是可以作为函数的返回值存在的

注意：不要返回局部变量引用

用法：函数调用作为左值

示例：

```
//返回局部变量引用
int& test01() {
    int a = 10; //局部变量
    return a;
}

//返回静态变量引用
int& test02() {
    static int a = 20;
    return a;
}

int main() {

    //不能返回局部变量的引用
    int& ref = test01();
    cout << "ref = " << ref << endl;
    cout << "ref = " << ref << endl;

    //如果函数做左值，那么必须返回引用
    int& ref2 = test02();
    cout << "ref2 = " << ref2 << endl;
    cout << "ref2 = " << ref2 << endl;

    test02() = 1000;

    cout << "ref2 = " << ref2 << endl;
    cout << "ref2 = " << ref2 << endl;

    system("pause");
}

return 0;
}
```

2.5 引用的本质

本质：引用的本质在c++内部实现是一个指针常量.

讲解示例：

```

//发现是引用，转换为 int* const ref = &a;
void func(int& ref){
    ref = 100; // ref是引用，转换为*ref = 100
}
int main(){
    int a = 10;

    //自动转换为 int* const ref = &a; 指针常量是指针指向不可改，也说明为什么引用不可更改
    int& ref = a;
    ref = 20; //内部发现ref是引用，自动帮我们转换为: *ref = 20;

    cout << "a:" << a << endl;
    cout << "ref:" << ref << endl;

    func(a);
    return 0;
}

```

结论：C++推荐用引用技术，因为语法方便，引用本质是指针常量，但是所有的指针操作编译器都帮我们做了

2.6 常量引用

作用：常量引用主要用来修饰形参，防止误操作

在函数形参列表中，可以加const修饰形参，防止形参改变实参

示例：

```

//引用使用的场景，通常用来修饰形参
void showValue(const int& v) {
    //v += 10;           //常量引用把所指向的对象看作常量，这里不能修改它指向的对象的值 会报错
    cout << v << endl;
}

int main() {

    //int& ref = 10;   引用本身需要一个合法的内存空间，因此这行错误
    //加入const就可以了，编译器优化代码，int temp = 10; const int& ref = temp;
    const int& ref = 10;

    //ref = 100; //加入const后不可以修改变量
    cout << ref << endl;

    //函数中利用常量引用防止误操作修改实参
    int a = 10;
    showValue(a);

    system("pause");

    return 0;
}

```

3 函数提高

3.1 函数默认参数

在C++中，函数的形参列表中的形参是可以有默认值的。

语法： 返回值类型 函数名 (参数= 默认值) {}

示例：

```
int func(int a, int b = 10, int c = 10) {
    return a + b + c;
}

//1. 如果某个位置参数有默认值，那么从这个位置往后，从左向右，必须都要有默认值，否则会报错
//2. 如果函数声明有默认值，函数实现的时候就不能有默认参数（声明和实现只能有一个有默认参数）
int func2(int a = 10, int b = 10);
//这里就是函数的声明，没有大括号就是，有时候会这样写先声明有这个函数，后面再写实现
int func2(int a, int b) {           //这里不能再写int a = 常数；因为编译器无法判断按照声明的来还是按照实现的
    return a + b;
}

int main() {

    cout << "ret = " << func(20, 20) << endl; //只要传一个以上即可，没传就用默认值
    cout << "ret = " << func(100) << endl;

    system("pause");

    return 0;
}
```

3.2 函数占位参数

C++中函数的形参列表里可以有占位参数，用来做占位，调用函数时必须填补该位置

语法： 返回值类型 函数名 (数据类型){}

在现阶段函数的占位参数存在意义不大，但是后面的课程中会用到该技术

示例：

```
//函数占位参数，占位参数也可以有默认参数
void func(int a, int) {
    cout << "this is func" << endl;
}

int main() {
    func(10, 10); //占位参数必须填补，否则会报错，当占位参数有默认参数时（比如int=10），那可以不填补
    system("pause");
    return 0;
}
```

3.3 函数重载

3.3.1 函数重载概述

作用：函数名可以相同，提高复用性

函数重载满足条件：

- 同一个作用域下（目前常写的不在main函数里的单个函数作用域是全局作用域）
- 函数名称相同
- 函数参数**类型不同** 或者 **个数不同** 或者 **顺序不同**

注意：函数的返回值不可以作为函数重载的条件

示例：

```

//函数重载需要函数都在同一个作用域下
void func()
{
    cout << "func 的调用! " << endl;
}
void func(int a)
{
    cout << "func (int a) 的调用! " << endl;
}
void func(double a)
{
    cout << "func (double a)的调用! " << endl;
}
void func(int a ,double b)
{
    cout << "func (int a ,double b) 的调用! " << endl;
}
void func(double a ,int b)
{
    cout << "func (double a ,int b)的调用! " << endl;
}

//函数返回值不同不可以作为函数重载条件
//int func(double a, int b)          //这里int 和上面的 void不能作为条件
//{
//    cout << "func (double a ,int b)的调用! " << endl;
//}

int main() {

    func();
    func(10);
    func(3.14);
    func(10,3.14);
    func(3.14 , 10);

    system("pause");
    return 0;
}

```

3.3.2 函数重载注意事项

- 引用作为重载条件
- 函数重载碰到函数默认参数

示例：

```

//函数重载注意事项
//1、引用作为重载条件的注意事项

void func(int &a)           //int &a =10 , 不合法，他只能调用一个变量
{
    cout << "func (int &a) 调用" << endl;
}

void func(const int &a)      // 只能传入一个常量
{
    cout << "func (const int &a) 调用" << endl;
}

//2、函数重载碰到函数默认参数的注意事项(写函数重载最好不要写默认参数，防止发生歧义)

void func2(int a, int b = 10)
{
    cout << "func2(int a, int b = 10) 调用" << endl;
}

void func2(int a)
{
    cout << "func2(int a) 调用" << endl;
}

int main() {
    int a = 10;
    func(a); //调用无const          func的两个重载函数类型不同,
    func(10); //调用有const

    //func2(10); //碰到默认参数产生歧义，需要避免，因为上述func2两个重载函数都只需要传入一个参数

    system("pause");
    return 0;
}

```

4 类和对象

C++面向对象的三大特性为：封装、继承、多态

C++认为万事万物都皆为对象，对象上有其属性和行为

例如：

人可以作为对象，属性有姓名、年龄、身高、体重...，行为有走、跑、跳、吃饭、唱歌...

车也可以作为对象，属性有轮胎、方向盘、车灯...，行为有载人、放音乐、放空调...

具有相同性质的对象，我们可以抽象称为类，人属于人类，车属于车类

4.1 封装

4.1.1 封装的意义

封装是C++面向对象三大特性之一

封装的意义：

- 将属性和行为作为一个整体，表现生活中的事物
- 将属性和行为加以权限控制

封装意义一：

在设计类的时候，属性和行为写在一起，表现事物

语法： class 类名 { 访问权限： 属性 / 行为 };

****示例1：** **设计一个圆类，求圆的周长

示例代码：

```

//圆周率
const double PI = 3.14;

//1、封装的意义
//将属性和行为作为一个整体，用来表现生活中的事物

//封装一个圆类，求圆的周长
//class代表设计一个类，后面跟着的是类名
class Circle
{
public: //访问权限 公共的权限

    //属性
    int m_r;//半径

    //行为 在类中通常用一个函数来实现
    //获取到圆的周长
    double calculateZC()
    {
        //2 * pi * r
        //获取圆的周长
        return 2 * PI * m_r;
    }
};

int main() {

    //通过圆类，创建圆的对象
    //实例化 (通过一个雷 创建一个对象的过程) c1就是一个具体的圆
    Circle c1;
    c1.m_r = 10; //给圆对象的半径 进行赋值操作

    //2 * pi * 10 == 62.8
    cout << "圆的周长为：" << c1.calculateZC() << endl;

    system("pause");

    return 0;
}

```

示例2：设计一个学生类，属性有姓名和学号，可以给姓名和学号赋值，可以显示学生的姓名和学号

示例2代码：

```

//学生类
class Student {
public:
    //类中的属性和行为 我们统一称为成员
    //属性 又称为成员属性 变量属性
    //行为 又称为成员函数 成员方法
    void setName(string name) {    //这里用行为给属性赋值也是可以的。也可以直接写属性，然后用class.属
        m_name = name;
    }
    void setID(int id) {
        m_id = id;
    }

    void showStudent() {
        cout << "name:" << m_name << " ID:" << m_id << endl;
    }
public:
    string m_name;
    int m_id;
};

int main() {

    Student stu;
    stu.setName("德玛西亚");
    stu.setID(250);
    stu.showStudent();

    system("pause");

    return 0;
}

```

封装意义二：

类在设计时，可以把属性和行为放在不同的权限下，加以控制

访问权限有三种：

1. public 公共权限
2. protected 保护权限
3. private 私有权限

示例：

```

//三种权限
//公共权限 public    类内可以访问  类外可以访问
//保护权限 protected  类内可以访问  类外不可以访问  (子类可以访问父类中的保护内容)
//私有权限 private   类内可以访问  类外不可以访问  (子类不可能访问父类的私有内容)  (这两个的区别在继承里比较)

class Person
{
    //姓名 公共权限
public:
    string m_Name;

    //汽车 保护权限
protected:
    string m_Car;

    //银行卡密码 私有权限
private:
    int m_Password;

public:
    void func()
    {
        m_Name = "张三";
        m_Car = "拖拉机";
        m_Password = 123456;
    }
};

int main() {

    Person p;
    p.m_Name = "李四";
    //p.m_Car = "奔驰"; //保护权限类外访问不到
    //p.m_Password = 123; //私有权限类外访问不到

    system("pause");

    return 0;
}

```

4.1.2 struct和class区别

在C++中 struct和class唯一的**区别**就在于 **默认的访问权限不同**

区别：

- struct 默认权限为公共
- class 默认权限为私有

```
class C1
{
    int m_A; //默认是私有权限
};

struct C2
{
    int m_A; //默认是公共权限
};

int main() {
    C1 c1;
    c1.m_A = 10; //错误，访问权限是私有，如果不加public默认是私有的不可访问的

    C2 c2;
    c2.m_A = 10; //正确，访问权限是公共

    system("pause");

    return 0;
}
```

4.1.3 成员属性设置为私有

**优点1： **将所有成员属性设置为私有，可以自己控制读写权限（用公开的行为间接访问和更改属性）

**优点2： **对于写权限，我们可以检测数据的有效性(防止数据超出有效范围)

示例：

```
class Person {
public:

    //姓名设置可读可写
    void setName(string name) {
        m_Name = name;
    }
    string getName()
    {
        return m_Name;
    }

    //获取年龄
    int getAge() {
        return m_Age;
    }
    //设置年龄
    void setAge(int age) {
        if (age < 0 || age > 150) {
            cout << "你个老妖精!" << endl;
            return;
        }
        m_Age = age;
    }

    //情人设置为只写
    void setLover(string lover) {
        m_Lover = lover;
    }

private:
    string m_Name; //可读可写 姓名

    int m_Age; //只读 年龄

    string m_Lover; //只写 情人
};

int main() {

    Person p;
    //姓名设置
    p.setName("张三");
    cout << "姓名: " << p.getName() << endl;

    //年龄设置
    p.setAge(50);
    cout << "年龄: " << p.getAge() << endl;

    //情人设置
    p.setLover("苍井");
    //cout << "情人: " << p.m_Lover << endl; //只写属性，不可以读取
```

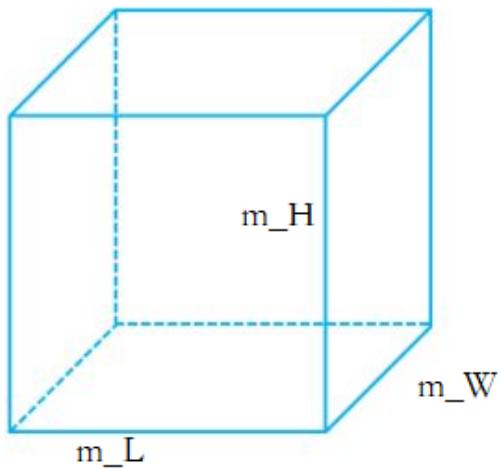
```
    system("pause");  
    return 0;  
}
```

练习案例1：设计立方体类

设计立方体类(Cube)

求出立方体的面积和体积

分别用全局函数和成员函数判断两个立方体是否相等。



```
#include <iostream>
#include <string>
using namespace std;

class Cube
{
public:
//设置获取长
    void setm_L(int L)
    {
        m_L = L;
    }
    int getm_L()
    {
        return m_L;
    }
//设置获取宽
    void setm_W(int W)
    {
        m_W = W;
    }
    int getm_W()
    {
        return m_W;
    }
//设置获取高
    void setm_H(int H)
    {
        m_H = H;
    }
    int getm_H()
    {
        return m_H;
    }
//获取面积
    int area()
    {
        return (m_L*m_W + m_W*m_H +m_L*m_H)*2;
    }
//获取体积
    int volume()
    {
        return(m_L*m_W*m_H);
    }
//成员函数判断两个立方体是否相等
bool IsSame( Cube &cube)
{
    if(getm_L() == cube.getm_L() && getm_W() == cube.getm_W() && getm_H() == cube.getm_H())
    {
        return true;
    }
    return false;
}
private:
    int m_L;
```

```

    int m_W;
    int m_H;

};

//全局函数判断两个立方体是否相同
bool IsSame(Cube &cube1, Cube &cube2)
{
    if(cube1.getm_L() == cube2.getm_L() && cube1.getm_W() == cube2.getm_W() && cube1.getm_H()
    {
        return true;
    }
    return false;
}

int main()
{
    Cube cube1;
    cube1.setm_L(10);
    cube1.setm_W(10);
    cube1.setm_H(10);
    Cube cube2;
    cube2.setm_L(10);
    cube2.setm_W(10);
    cube2.setm_H(10);

    //bool ret = IsSame(cube1,cube2);      //全局函数判断两个立方体是否相等
    bool ret = cube1.Issame(cube2);          //成员函数判断两个立方体是否相等
    if(ret == false)
    {
        cout <<"两个立方体不相等" << endl;
    }
    else
    {
        cout << "两个立方体相等" << endl;
    }

    cout << cube1.area() << endl;
    cout << cube1.volume() << endl;
}

```

练习案例2：点和圆的关系

设计一个圆形类（Circle），和一个点类（Point），计算点和圆的关系。

本练习了解1.(在类中可以让另一个类作为本类的成员)

2.将不同类拆解成不同文件



我们都会有什么关系?

```
#include <iostream>
#include <string>
using namespace std;

//点的类
class Point
{
public:
//设置访问x坐标
    void setm_x(int x)
    {
        m_x = x;
    }
    int getm_x()
    {
        return m_x;
    }
//设置访问y坐标
    void setm_y(int y)
    {
        m_y = y;
    }
    int getm_y()
    {
        return m_y;
    }
private:
    int m_x;
    int m_y;
};

//圆的类
class Circle
{
public:
//设置访问圆的半径
    void setm_r(int r)
    {
        m_r = r;
    }
    int getm_r()
    {
        return m_r;
    }
//设置访问圆的圆点
    void setm_center(Point center)
    {
        m_center = center;
    }
    Point getm_center()
    {
        return m_center;
    }
private:
    int m_r;
```

```

    Point m_center;
};

//判断点与圆的位置关系
void PositionRelation(Circle &c, Point &p )
{

    int distance = (c.getm_center().getm_x() - p.getm_x())*(c.getm_center().getm_x() - p.getm_x())
    + (c.getm_center().getm_y() - p.getm_y())*(c.getm_center().getm_y() - p.getm_y());
    cout<<distance<<endl;
    int RR = c.getm_r()*c.getm_r();
    //千万要注意在c++中幂函数不能写成x^2这种，这里卡了一个小时。可以用math.h的库函数pow；格式：int pow(x,y);

    if (distance == RR)
    {
        cout << "点在圆上" << endl;
    }
    if (distance > RR)
    {
        cout << "点在圆外" << endl;
    }
    if (distance < RR)
    {
        cout << "点在圆内" << endl;
    }
}
int main()
{
    Point p1;
    p1.setm_x(10);
    p1.setm_y(10);
    Point p2;
    p2.setm_x(10);
    p2.setm_y(0);
    Circle c1;
    c1.setm_center(p2);
    c1.setm_r(10);

    PositionRelation(c1,p1);

    return 0;
}

```

将不同类拆解成不同文件

这里一共有五个文件分别是1.point_circle.cpp 2.circle.h 3.circle.cpp 4.point.h
5.point.cpp

```

#include <iostream>
#include <string>
#include "point.h"
#include "circle.h"
using namespace std;

//判断点与圆的位置关系
void PositionRelation(Circle &c, Point &p )
{
    int distance = (c.getm_center().getm_x() - p.getm_x())*(c.getm_center().getm_x() - p.getm_x()
    + (c.getm_center().getm_y() - p.getm_y())*(c.getm_center().getm_y() - p.getm_y());
    cout<<distance<<endl;
    int RR = c.getm_r()*c.getm_r();
    //千万要注意在c++中幂函数不能写成x^2这种，这里卡了一个小时。可以用math.h的库函数pow；格式：int pow(x,y);

    if (distance == RR)
    {
        cout << "点在圆上" << endl;
    }
    if (distance > RR)
    {
        cout << "点在圆外" << endl;
    }
    if (distance < RR)
    {
        cout << "点在圆内" << endl;
    }
}
int main()
{
    Point p1;
    p1.setm_x(10);
    p1.setm_y(10);
    Point p2;
    p2.setm_x(10);
    p2.setm_y(0);
    Circle c1;
    c1.setm_center(p2);
    c1.setm_r(10);

    PositionRelation(c1,p1);

    return 0;
}

```

```
#pragma once
#include <iostream>
#include "point.h"
using namespace std;

//圆的类
class Circle
{
public:
//设置访问圆的半径
void setm_r(int r);

int getm_r();

//设置访问圆的圆点
void setm_center(Point center);

Point getm_center();

private:
int m_r;
Point m_center;
};

#endif
```

```
#pragma once
#include <iostream>
#include "point.h"
using namespace std;

//圆的类
class Circle
{
public:
//设置访问圆的半径
void setm_r(int r);

int getm_r();

//设置访问圆的圆点
void setm_center(Point center);

Point getm_center();

private:
int m_r;
Point m_center;
};
```

```
#include "circle.h"

//设置访问圆的半径
void Circle::setm_r(int r)
{
    m_r = r;
}
int Circle::getm_r()
{
    return m_r;
}
//设置访问圆的圆点
void Circle::setm_center(Point center)
{
    m_center = center;
}
Point Circle::getm_center()
{
    return m_center;
}

#pragma once
#include <iostream>
using namespace std;

class Point
{
public:
//设置访问x坐标
    void setm_x(int x);

    int getm_x();

//设置访问y坐标
    void setm_y(int y);

    int getm_y();

private:
    int m_x;
    int m_y;
};
```

```
#include "point.h"

//只需要留下成员函数的实现即可
void Point::setm_x(int x)
{
    m_x = x;
}
int Point::getm_x()
{
    return m_x;
}

void Point::setm_y(int y)
{
    m_y = y;
}
int Point::getm_y()
{
    return m_y;
}
```

运行时可对源文件进行编译生成可执行文件。

4.2 对象的初始化和清理

- 生活中我们买的电子产品都基本会有出厂设置，在某一天我们不用时候也会删除一些自己信息数据保证安全
- C++中的面向对象来源于生活，每个对象也都会有初始设置以及 对象销毁前的清理数据的设置。

4.2.1 构造函数和析构函数

对象的**初始化和清理**也是两个非常重要的安全问题

一个对象或者变量没有初始状态，对其使用后果是未知

同样的使用完一个对象或变量，没有及时清理，也会造成一定的安全问题

c++利用了**构造函数**和**析构函数**解决上述问题，这两个函数将会被编译器自动调用，完成对象初始化和清理工作。

对象的初始化和清理工作是编译器强制要我们做的事情，因此如果**我们不提供构造和析构，编译器会提供编译器提供的构造函数和析构函数是空实现。**

- 构造函数：主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用。
- 析构函数：主要作用在于对象**销毁前**系统自动调用，执行一些清理工作。

构造函数语法： 类名(){}
1. 构造函数，没有返回值也不写void

2. 函数名称与类名相同
3. 构造函数可以有参数，因此可以发生重载
4. 程序在调用对象时候会自动调用构造，无须手动调用，而且只会调用一次

析构函数语法： ~类名(){}

1. 析构函数，没有返回值也不写void
2. 函数名称与类名相同，在名称前加上符号 ~
3. 析构函数不可以有参数，因此不可以发生重载
4. 程序在对象销毁前会自动调用析构，无须手动调用，而且只会调用一次

```
#include <iostream>
using namespace std;

//对象的初始化和清理
//1、构造函数 进行初始化操作
class Person
{
public:
    Person()
    {
        cout << "这是一个构造函数" << endl;
    }
    ~Person()
    {
        cout << "这是一个析构函数的调用" << endl;
    }
//构造函数和析构函数都是必须有的实现，如果我们自己不提供，编译器会提供一个空实现的构造和析构
};

void test01()
{
    Person p;           //在栈上的数据，test01执行完毕后，释放这个对象
}

int main()
{
    Person p1;
//如果是Windows系统用system ("pause") 可以看到系统没有调用析构函数，因为这个main函数不执行完毕不会释放栈区|
// //test01();

    return 0;
}
```

4.2.2 构造函数的分类及调用

两种分类方式：

按参数分为：有参构造和无参构造（默认构造，默认构造的函数无参数）

按类型分为： 普通构造和拷贝构造

三种调用方式：

括号法

显示法

隐式转换法

示例：

```

//1、构造函数分类
// 按照参数分类分为 有参和无参构造    无参又称为默认构造函数
// 按照类型分类分为 普通构造和拷贝构造

#include <iostream>
using namespace std;

class Person
{
public:
    Person()
    {
        cout << "这是一个无参构造函数" << endl;
    }
    Person(int a)
    {
        age = a;
        cout << "这是一个有参构造函数" << endl;
    }
    //拷贝构造函数
    Person(const Person &p)
    {
        //将传入的人身上的所有属性, copy到我身上, 记住这个结构, 用到const是因为这个不能随意改变。
        age = p.age;
        cout << "这是一个拷贝构造" << endl;
    }
    ~Person()
    {
        cout << "这是一个析构函数!" << endl;
    }
    int age;
};

void test02()
{
    // //1.括号法
    // Person p1;           //如果不输入参数, 默认构造函数调用无参构造函数
    // Person p2(10);       //传入参数就会调用有参构造函数
    // Person p3(p2);       //当传入一个当前类的对象时, 调用拷贝构造函数
    // cout << "p2的年龄为:" << p2.age << endl;
    // cout << "p3的年龄为:" << p3.age << endl;      //这里p3拷贝的p2的属性, 所以年龄也为10;
    // //注意调用构造函数时, 不要加(); 比如上面p1后加()写成Person p1();会被系统认为是一个函数的声明

    //2.显示法
    // Person p1;
    // Person p2 = Person(10); //有参构造
    // Person p3 = Person(p2); //拷贝构造
    // Person(10);           //匿名对象 特点: 当前行执行结束后, 系统会立即回收掉匿名对象
    // 注意事项2 不要利用拷贝构造函数初始化匿名对象
    // Person(p3);          //编译器会认为这是一个对象声明 和上面的Person p3(p2)重复了

    //3.隐式转换法
}

```

```
Person p4 = 10; //相当于写了Person p4 = Person(10);
Person p5 = p4; //拷贝构造

}

int main()
{
    test02();
    return 0;
}
```

4.2.3 拷贝构造函数调用时机

C++中拷贝构造函数调用时机通常有三种情况

- 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

示例：

```

class Person {
public:
    Person() {
        cout << "无参构造函数!" << endl;
        mAge = 0;
    }
    Person(int age) {
        cout << "有参构造函数!" << endl;
        mAge = age;
    }
    Person(const Person& p) {
        cout << "拷贝构造函数!" << endl;
        mAge = p.mAge;
    }
    //析构函数在释放内存之前调用
    ~Person() {
        cout << "析构函数!" << endl;
    }
public:
    int mAge;
};

//1. 使用一个已经创建完毕的对象来初始化一个新对象
void test01() {

    Person man(100); //p对象已经创建完毕
    Person newman(man); //调用拷贝构造函数
    Person newman2 = man; //拷贝构造

    //Person newman3;
    //newman3 = man; //不是调用拷贝构造函数，赋值操作
}

//2. 值传递的方式给函数参数传值
//相当于Person p1 = p;
void doWork(Person p1) {}
void test02() {
    Person p; //无参构造函数
    doWork(p); //这里会调用拷贝构造函数，因为值传递的过程其实就是copy出一份放在内存里，地址传递不会co
}

//3. 以值方式返回局部对象
Person doWork2()
{
    Person p1;
    cout << (int *)&p1 << endl;
    return p1;           //这里会调用拷贝函数 因为上面的p1是局部变量
}

void test03()
{
    Person p = doWork2();
    cout << (int *)&p << endl;
}

```

```
int main() {  
    //test01();  
    //test02();  
    test03();  
  
    system("pause");  
  
    return 0;  
}
```

4.2.4 构造函数调用规则

默认情况下，c++编译器至少给一个类添加3个函数

1. 默认构造函数(无参， 函数体为空)
2. 默认析构函数(无参， 函数体为空)
3. 默认拷贝构造函数， 对属性进行值拷贝

构造函数调用规则如下：

- 如果用户定义有参构造函数，c++不在提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数，c++不会再提供其他构造函数

示例：

```
class Person {
public:
    //无参（默认）构造函数
    Person() {
        cout << "无参构造函数!" << endl;
    }
    //有参构造函数
    Person(int a) {
        age = a;
        cout << "有参构造函数!" << endl;
    }
    //拷贝构造函数
    Person(const Person& p) {
        age = p.age;
        cout << "拷贝构造函数!" << endl;
    }
    //析构函数
    ~Person() {
        cout << "析构函数!" << endl;
    }
public:
    int age;
};

void test01()
{
    Person p1(18);
    //如果不写拷贝构造，编译器会自动添加拷贝构造，并且做浅拷贝操作（所有的属性都做了赋值操作）
    Person p2(p1);

    cout << "p2的年龄为：" << p2.age << endl;
}

void test02()
{
    //如果用户提供有参构造，编译器不会提供默认构造，会提供拷贝构造
    Person p1; //此时如果用户自己没有提供默认构造（提供了有参构造），会出错
    Person p2(10); //用户提供的有参
    Person p3(p2); //此时如果用户没有提供拷贝构造，编译器会提供

    //如果用户提供拷贝构造，编译器不会提供其他构造函数
    Person p4; //此时如果用户自己没有提供默认构造，会出错
    Person p5(10); //此时如果用户自己没有提供有参，会出错
    Person p6(p5); //用户自己提供拷贝构造
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

4.2.5 深拷贝与浅拷贝

深浅拷贝是面试经典问题，也是常见的一个坑

浅拷贝：简单的赋值拷贝操作

深拷贝：**在堆区重新申请空间，进行拷贝操作**

示例：

```
class Person {
public:
    //无参（默认）构造函数
    Person() {
        cout << "无参构造函数!" << endl;
    }
    //有参构造函数
    Person(int age ,int height) {

        cout << "有参构造函数!" << endl;

        m_age = age;
        m_height = new int(height);

    }
    //拷贝构造函数
    Person(const Person& p) {
        cout << "拷贝构造函数!" << endl;
        //如果不利用深拷贝在堆区创建新内存，会导致浅拷贝带来的重复释放堆区问题
        m_age = p.m_age;
        m_height = p.m_height; //浅拷贝 在多次调用析构函数释放数据时会报错。
        m_height = new int(*p.m_height);

    }

    //析构函数
    ~Person() {
        cout << "析构函数!" << endl;
        //析构代码，将对去开辟的数据做释放
        if (m_height != NULL)
        {
            delete m_height;
        }
    }
public:
    int m_age;
    int* m_height;
};

void test01()
{
    Person p1(18, 180);

    Person p2(p1);

    cout << "p1的年龄: " << p1.m_age << " 身高: " << *p1.m_height << endl;

    cout << "p2的年龄: " << p2.m_age << " 身高: " << *p2.m_height << endl;
}

int main() {

    test01();

    system("pause");
}
```

```
    return 0;
}
```

总结：如果属性有在堆区开辟的，一定要自己提供拷贝构造函数，防止浅拷贝带来的问题

4.2.6 初始化列表

作用：

C++提供了初始化列表语法，用来初始化属性

语法：构造函数()：属性1(值1), 属性2 (值2) ... {}

示例：

```
class Person {
public:
    //传统方式初始化（构造函数做初始化）
    //Person(int a, int b, int c) {
    //    m_A = a;
    //    m_B = b;
    //    m_C = c;
    //}

    //初始化列表方式初始化
    Person(int a, int b, int c) :m_A(a), m_B(b), m_C(c) {}
    void PrintPerson() {
        cout << "mA:" << m_A << endl;
        cout << "mB:" << m_B << endl;
        cout << "mC:" << m_C << endl;
    }
private:
    int m_A;
    int m_B;
    int m_C;
};

int main() {
    Person p(1, 2, 3);
    p.PrintPerson();

    system("pause");
    return 0;
}
```

4.2.7 类对象作为类成员

C++类中的成员可以是另一个类的对象，我们称该成员为 对象成员

例如：

```
class A {}  
class B  
{  
    A a;  
}
```

B类中有对象A作为成员， A为对象成员

那么当创建B对象时， A与B的构造和析构的顺序是谁先谁后？(这有什么实际应用呢)

示例：

```
class Phone
{
public:
    Phone(string name)
    {
        m_PhoneName = name;
        cout << "Phone构造" << endl;
    }

    ~Phone()
    {
        cout << "Phone析构" << endl;
    }

    string m_PhoneName;
};

class Person
{
public:
    //初始化列表可以告诉编译器调用哪一个构造函数
    Person(string name, string pName) :m_Name(name), m_Phone(pName)
    {
        cout << "Person构造" << endl;
    }

    ~Person()
    {
        cout << "Person析构" << endl;
    }

    void playGame()
    {
        cout << m_Name << " 使用" << m_Phone.m_PhoneName << " 牌手机！" << endl;
    }

    string m_Name;
    Phone m_Phone;
};

void test01()
{
    //当类中成员是其他类对象时，我们称该成员为 对象成员
    //构造的顺序是：先调用对象成员的构造，再调用本类构造
    //析构顺序与构造相反
    Person p("张三", "苹果X");
    p.playGame();
}

int main() {
```

```
    test01();  
  
    system("pause");  
  
    return 0;  
}
```

4.2.8 静态成员

静态成员就是在成员变量和成员函数前加上关键字static，称为静态成员

静态成员分为：

- 静态成员变量
 - 所有对象共享同一份数据
 - 在编译阶段分配内存
 - 类内声明，类外初始化
- 静态成员函数
 - 所有对象共享同一个函数？？？
 - 静态成员函数只能访问静态成员变量

示例1：静态成员变量

```

class Person
{
public:
    static int m_A; //静态成员变量

    //静态成员变量特点:
    //1 在编译阶段分配内存
    //2 类内声明, 类外初始化
    //3 所有对象共享同一份数据

private:
    static int m_B; //静态成员变量也是有访问权限的
};

int Person::m_A = 10;
int Person::m_B = 10;

void test01()
{
    //静态成员变量两种访问方式

    //1、通过对象
    Person p1;
    p1.m_A = 100;
    cout << "p1.m_A = " << p1.m_A << endl;

    Person p2;
    p2.m_A = 200;
    cout << "p1.m_A = " << p1.m_A << endl; //共享同一份数据, 所以这里会打印200
    cout << "p2.m_A = " << p2.m_A << endl;

    //2、通过类名
    cout << "m_A = " << Person::m_A << endl;

    //cout << "m_B = " << Person::m_B << endl; //私有权限访问不到
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

示例2：静态成员函数

```
class Person
{
public:
    //静态成员函数特点:
    //1 程序共享一个函数
    //2 静态成员函数只能访问静态成员变量

    static void func()
    {
        cout << "func调用" << endl;
        m_A = 100; //共享数据 并不需要实例化对象
        //m_B = 100; //错误，不可以访问非静态成员变量
    }

    static int m_A; //静态成员变量
    int m_B; //

private:
    //静态成员函数也是有访问权限的
    static void func2()
    {
        cout << "func2调用" << endl;
    }
};

int Person::m_A = 10;

void test01()
{
    //静态成员函数两种访问方式

    //1、通过对象
    Person p1;
    p1.func();

    //2、通过类名
    Person::func();

    //Person::func2(); //私有权限，类外访问不到
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

4.3 C++对象模型和this指针

4.3.1 成员变量和成员函数分开存储

在C++中，类内的成员变量和成员函数分开存储

只有非静态成员变量才属于类的对象上

```
class Person {
public:
    Person() {
        mA = 0;
    }
    //非静态成员变量占对象空间
    int mA;
    //静态成员变量不占对象空间
    static int mB;
    //函数也不占对象空间，所有函数共享一个函数实例
    void func() {
        cout << "mA:" << this->mA << endl;
    }
    //静态成员函数也不占对象空间
    static void sfunc() {
    }
};

int main() {
    cout << sizeof(Person) << endl;//空对象占一个字节，上述不是空对象，有一个非静态成员int，因此分配
    //c++编译器为每个空对象也分配一个字节空间，是为了区分空对象占内存的位置
    //每个空对象也应该有一个独一无二的内存地址

    system("pause");
    return 0;
}
```

4.3.2 this指针概念

通过4.3.1我们知道在C++中成员变量和成员函数是分开存储的

每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码

那么问题是：这一块代码是如何区分那个对象调用自己的呢？

C++通过提供特殊的对象指针，this指针，解决上述问题。**this指针指向被调用的成员函数所属的对象**

this指针是隐含每一个非静态成员函数内的一种指针

this指针不需要定义，直接使用即可

this指针的用途：

- 当形参和成员变量同名时，可用this指针来区分
- 在类的非静态成员函数中返回对象本身，可使用return *this

```
#include <iostream>
using namespace std;

class Person
{
public:

    Person(int age)
    {
        //1、当形参和成员变量同名时，可用this指针来区分

        this->age = age;
    }

    Person& PersonAddPerson(Person p)
    //返回对象本身要用返回引用的方式，如果这里返回Person是返回值的方式，相当于创建了一个新对象
    {
        this->age += p.age;
        //返回对象本身
        return *this;
    }

    int age;
};

void test01()
{
    Person p1(10);
    cout << "p1.age = " << p1.age << endl;
    // this指针指向 被调用的成员函数 所属的对象

    Person p2(10);
    p2.PersonAddPerson(p1).PersonAddPerson(p1).PersonAddPerson(p1);
    //这里函数调用之后返回的是对象本身，因此可以继续调用函数
    cout << "p2.age = " << p2.age << endl; //cout也是调用之后可以追加调用
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

4.3.3 空指针访问成员函数

C++中空指针也是可以调用成员函数的，但是也要注意有没有用到this指针

如果用到this指针，需要加以判断保证代码的健壮性

示例：

```
//空指针访问成员函数
class Person {
public:

    void ShowClassName() {
        cout << "我是Person类!" << endl;
    }

    void ShowPerson() {
        if (this == NULL) {
            return;
        }
        cout << mAge << endl; //每一个函数调用的成员变量前面都默认有this->指针，只是被省略了。
    }

public:
    int mAge;
};

void test01()
{
    Person * p = NULL;
    p->ShowClassName(); //空指针，可以调用成员函数
    p->ShowPerson(); //但是如果成员函数中用到了this指针，就不可以了
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

4.3.4 const修饰成员函数

常函数：

- 成员函数后加const后我们称为这个函数为**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

常对象：

- 声明对象前加const称该对象为常对象
- 常对象只能调用常函数

示例：

```
class Person {
public:
    Person() {
        m_A = 0;
        m_B = 0;
    }

    //this指针的本质是一个指针常量，指针的指向不可修改
    //如果想让指针指向的值也不可以修改，需要声明常函数
    void ShowPerson() const {
        //const Type* const pointer;
        //this = NULL; //不能修改指针的指向 Person* const this;
        //this->mA = 100; //但是this指针指向的对象的数据是可以修改的

        //const修饰成员函数，表示指针指向的内存空间的数据不能修改，除了mutable修饰的变量
        this->m_B = 100;
    }

    void MyFunc() const {
        //mA = 10000;
    }
}

public:
    int m_A;
    mutable int m_B; //可修改 可变的
};

//const修饰对象 常对象
void test01() {

    const Person person; //常量对象
    cout << person.m_A << endl;
    //person.mA = 100; //常对象不能修改成员变量的值，但是可以访问
    person.m_B = 100; //但是常对象可以修改mutable修饰成员变量

    //常对象访问成员函数
    person.MyFunc();
    //常对象只能调用常函数。（因为普通函数可以修改成员属性，所以常对象不能调用普通成员函数）

}

int main() {
    test01();

    system("pause");
    return 0;
}
```

4.4 友元

生活中你的家有客厅(Public)，有你的卧室(Private)

客厅所有来的客人都可以进去，但是你的卧室是私有的，也就是说只有你能进去

但是呢，你也可以允许你的好闺蜜好基友进去。

在程序里，有些私有属性 也想让类外特殊的一些函数或者类进行访问，就需要用到友元的技术

友元的目的就是让一个函数或者类 访问另一个类中私有成员

友元的关键字为 `friend`

友元的三种实现

- 全局函数做友元
- 类做友元
- 成员函数做友元

4.4.1 全局函数做友元

```
class Building
{
    //告诉编译器 goodGay全局函数 是 Building类的好朋友，可以访问类中的私有内容
    friend void goodGay(Building * building); // (格式：friend+全局函数的声明)

public:

    Building()
    {
        this->m_SittingRoom = "客厅";
        this->m_BedRoom = "卧室";
    }

public:
    string m_SittingRoom; //客厅

private:
    string m_BedRoom; //卧室
};

void goodGay(Building * building)
{
    cout << "好基友正在访问：" << building->m_SittingRoom << endl;
    cout << "好基友正在访问：" << building->m_BedRoom << endl;
}

void test01()
{
    Building b;
    goodGay(&b);
}

int main(){
    test01();

    system("pause");
    return 0;
}
```

4.4.2 类做友元

```
class Building;
class goodGay
{
public:
    goodGay();
    void visit();

private:
    Building *building;
};

class Building
{
    //告诉编译器 goodGay类是Building类的好朋友，可以访问到Building类中私有内容
    friend class goodGay;

public:
    Building();

public:
    string m_SittingRoom; //客厅
private:
    string m_BedRoom;//卧室
};

Building::Building()           //也可以类外写成员函数的实现
{
    this->m_SittingRoom = "客厅";
    this->m_BedRoom = "卧室";
}

goodGay::goodGay()
{
    building = new Building;
}

void goodGay::visit()
{
    cout << "好基友正在访问" << building->m_SittingRoom << endl;
    cout << "好基友正在访问" << building->m_BedRoom << endl;
}

void test01()
{
    goodGay gg;
    gg.visit();
}

int main(){
```

```
    test01();  
  
    system("pause");  
    return 0;  
}
```

4.4.3 成员函数做友元

```
class Building;
class goodGay
{
public:
    goodGay();
    void visit(); //只让visit函数作为Building的好朋友，可以访问Building中私有内容
    void visit2();

private:
    Building *building;
};

class Building
{
    //告诉编译器 goodGay类中的visit成员函数 是Building好朋友，可以访问私有内容
    friend void goodGay::visit();

public:
    Building();
    string m_SittingRoom; //客厅
private:
    string m_BedRoom; //卧室
};

Building::Building()
{
    this->m_SittingRoom = "客厅";
    this->m_BedRoom = "卧室";
}

goodGay::goodGay()
{
    building = new Building;
}

void goodGay::visit()
{
    cout << "好基友正在访问" << building->m_SittingRoom << endl;
    cout << "好基友正在访问" << building->m_BedRoom << endl;
}

void goodGay::visit2()
{
    cout << "好基友正在访问" << building->m_SittingRoom << endl;
    //cout << "好基友正在访问" << building->m_BedRoom << endl;
}

void test01()
```

```
{  
    goodGay gg;  
    gg.visit();  
  
}  
  
int main(){  
  
    test01();  
  
    system("pause");  
    return 0;  
}
```

4.5 运算符重载

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

4.5.1 加号运算符重载

作用：实现两个自定义数据类型相加的运算

```
class Person {
public:
    Person() {};
    Person(int a, int b)
    {
        this->m_A = a;
        this->m_B = b;
    }
    //成员函数实现 + 号运算符重载
    Person operator+(const Person& p) {      //operator+是c++定义的函数名
        Person temp;
        temp.m_A = this->m_A + p.m_A;
        temp.m_B = this->m_B + p.m_B;
        return temp;
    }

public:
    int m_A;
    int m_B;
};

//全局函数实现 + 号运算符重载
//Person operator+(const Person& p1, const Person& p2) {
//    Person temp(0, 0);
//    temp.m_A = p1.m_A + p2.m_A;
//    temp.m_B = p1.m_B + p2.m_B;
//    return temp;
//}

//运算符重载 可以发生函数重载
Person operator+(const Person& p2, int val)
{
    Person temp;
    temp.m_A = p2.m_A + val;
    temp.m_B = p2.m_B + val;
    return temp;
}

void test() {

    Person p1(10, 10);
    Person p2(20, 20);

    //成员函数方式
    Person p3 = p2 + p1; //相当于 p2.operator+(p1) (当使用编译器定义的这种函数名后即可简化成+)
    cout << "mA:" << p3.m_A << " mB:" << p3.m_B << endl;

    //全局函数方式
    Person p4 = p3 + 10; //相当于 operator+(p3,10)
    cout << "mA:" << p4.m_A << " mB:" << p4.m_B << endl;

}

int main() {
```

```
    test();  
  
    system("pause");  
  
    return 0;  
}
```

| 总结1：对于内置的数据类型的表达式的的运算符是不可能改变的（比如 $1+1=2$ 不能乱改）

| 总结2：不要滥用运算符重载（相加实现相减等等，会让别人看不懂自己代码）

4.5.2 左移运算符重载

作用：可以输出自定义数据类型

```

class Person {
    friend ostream& operator<<(ostream& out, Person& p);

public:

    Person(int a, int b)
    {
        this->m_A = a;
        this->m_B = b;
    }

    //成员函数 实现不了 p << cout 不是我们想要的效果
    //void operator<<(Person& p){
    //}

private:
    int m_A;
    int m_B;
};

//全局函数实现左移重载
//ostream对象只能有一个
ostream& operator<<(ostream& out, Person& p) {
    out << "a:" << p.m_A << " b:" << p.m_B;
    return out;
}

void test() {

    Person p1(10, 20);

    cout << p1 << "hello world" << endl; //链式编程
}

int main() {

    test();

    system("pause");

    return 0;
}

```

总结：重载左移运算符配合友元可以实现输出自定义数据类型

4.5.3 递增运算符重载

作用：通过重载递增运算符，实现自己的整型数据

```
class MyInteger {

    friend ostream& operator<<(ostream& out, MyInteger myint);

public:
    MyInteger() {
        m_Num = 0;
    }
    //前置++
    MyInteger& operator++() {
        //先++
        m_Num++;
        //再返回
        return *this;
    }

    //后置++
    MyInteger operator++(int) {
        //先返回
        MyInteger temp = *this; //记录当前本身的值，然后让本身的值加1，但是返回的是以前的值，达?
        m_Num++;
        return temp;
    }

private:
    int m_Num;
};

ostream& operator<<(ostream& out, MyInteger myint) {
    out << myint.m_Num;
    return out;
}

//前置++ 先++ 再返回
void test01() {
    MyInteger myInt;
    cout << ++myInt << endl;
    cout << myInt << endl;
}

//后置++ 先返回 再++
void test02() {

    MyInteger myInt;
    cout << myInt++ << endl;
    cout << myInt << endl;
}

int main() {

    test01();
    //test02();
}
```

```
    system("pause");  
  
    return 0;  
}
```

| 总结：前置递增返回引用，后置递增返回值

4.5.4 赋值运算符重载

C++编译器至少给一个类添加4个函数

1. 默认构造函数(无参，函数体为空)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对属性进行值拷贝
4. 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

示例：

```
class Person
{
public:

    Person(int age)
    {
        //将年龄数据开辟到堆区
        m_Age = new int(age);
    }

    //重载赋值运算符
    Person& operator=(Person &p)
    {
        //应该先判断是否有属性在堆区，如果有删除
        if (m_Age != NULL)
        {
            delete m_Age;
            m_Age = NULL;
        }
        //编译器提供的代码是浅拷贝
        //m_Age = p.m_Age;

        //提供深拷贝 解决浅拷贝的问题
        m_Age = new int(*p.m_Age);

        //返回自身
        return *this;
    }

    ~Person()
    {
        if (m_Age != NULL)
        {
            delete m_Age;
            m_Age = NULL;
        }
    }

    //年龄的指针
    int *m_Age;
};

void test01()
{
    Person p1(18);

    Person p2(20);

    Person p3(30);

    p3 = p2 = p1; //赋值操作
```

```
cout << "p1的年龄为：" << *p1.m_Age << endl;
cout << "p2的年龄为：" << *p2.m_Age << endl;
cout << "p3的年龄为：" << *p3.m_Age << endl;
}

int main() {
    test01();

    //int a = 10;
    //int b = 20;
    //int c = 30;

    //c = b = a;
    //cout << "a = " << a << endl;
    //cout << "b = " << b << endl;
    //cout << "c = " << c << endl;

    system("pause");
    return 0;
}
```

4.5.5 关系运算符重载

作用：重载关系运算符，可以让两个自定义类型对象进行对比操作

示例：

```
class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }

    bool operator==(Person & p)
    {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    bool operator!=(Person & p)
    {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
        {
            return false;
        }
        else
        {
            return true;
        }
    }

    string m_Name;
    int m_Age;
};

void test01()
{
    //int a = 0;
    //int b = 0;

    Person a("孙悟空", 18);
    Person b("孙悟空", 18);

    if (a == b)
    {
        cout << "a和b相等" << endl;
    }
    else
    {
        cout << "a和b不相等" << endl;
    }

    if (a != b)
```

```
{  
    cout << "a和b不相等" << endl;  
}  
else  
{  
    cout << "a和b相等" << endl;  
}  
}
```

```
int main() {  
  
    test01();  
  
    system("pause");  
  
    return 0;  
}
```

4.5.6 函数调用运算符重载

- 函数调用运算符 () 也可以重载
- 由于重载后使用的方式非常像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

示例：

```

class MyPrint
{
public:
    void operator()(string text)
    {
        cout << text << endl;
    }

};

void test01()
{
    //重载的 () 操作符 也称为仿函数
    MyPrint myFunc;
    myFunc("hello world");
}

class MyAdd
{
public:
    int operator()(int v1, int v2)
    {
        return v1 + v2;
    }

};

void test02()
{
    MyAdd add;
    int ret = add(10, 10);
    cout << "ret = " << ret << endl;

    //匿名对象调用
    cout << "MyAdd()(100,100) = " << MyAdd()(100, 100) << endl;
}

int main() {
    test01();
    test02();

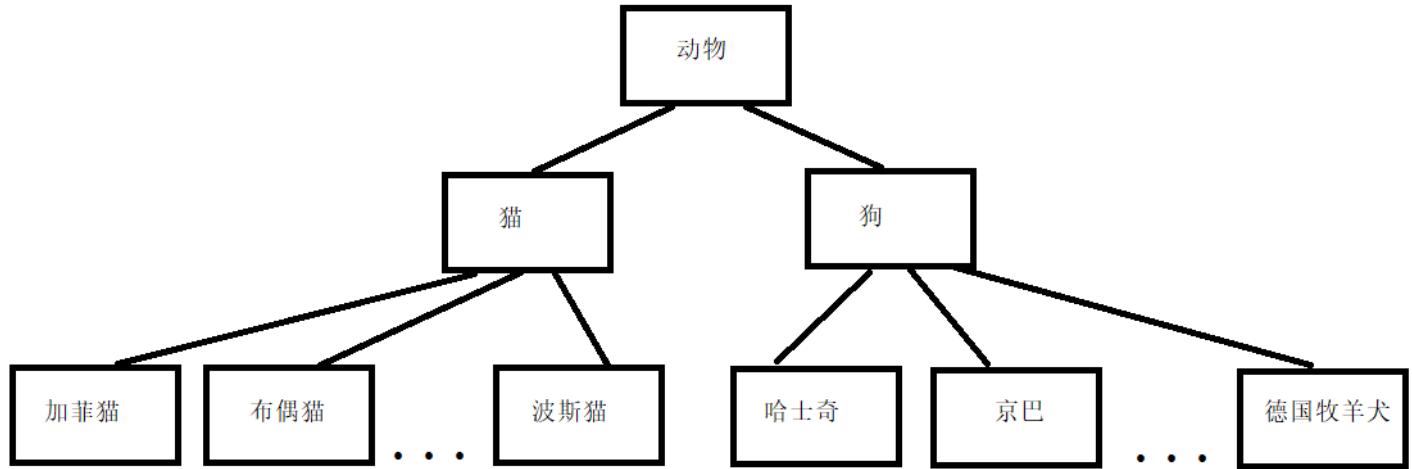
    system("pause");
    return 0;
}

```

4.6 继承

继承是面向对象三大特性之一

有些类与类之间存在特殊的关系，例如下图中：



我们发现，定义这些类时，下级别的成员除了拥有上一级的共性，还有自己的特性。

这个时候我们就可以考虑利用继承的技术，减少重复代码

4.6.1 继承的基本语法

例如我们看到很多网站中，都有公共的头部，公共的底部，甚至公共的左侧列表，只有中心内容不同

接下来我们分别利用普通写法和继承的写法来实现网页中的内容，看一下继承存在的意义以及好处

普通实现：

```
//Java页面
class Java
{
public:
    void header()
    {
        cout << "首页、公开课、登录、注册... (公共头部)" << endl;
    }
    void footer()
    {
        cout << "帮助中心、交流合作、站内地图...(公共底部)" << endl;
    }
    void left()
    {
        cout << "Java,Python,C++...(公共分类列表)" << endl;
    }
    void content()
    {
        cout << "JAVA学科视频" << endl;
    }
};

//Python页面
class Python
{
public:
    void header()
    {
        cout << "首页、公开课、登录、注册... (公共头部)" << endl;
    }
    void footer()
    {
        cout << "帮助中心、交流合作、站内地图...(公共底部)" << endl;
    }
    void left()
    {
        cout << "Java,Python,C++...(公共分类列表)" << endl;
    }
    void content()
    {
        cout << "Python学科视频" << endl;
    }
};

//C++页面
class CPP
{
public:
    void header()
    {
        cout << "首页、公开课、登录、注册... (公共头部)" << endl;
    }
    void footer()
    {
        cout << "帮助中心、交流合作、站内地图...(公共底部)" << endl;
    }
    void left()
```

```

{
    cout << "Java,Python,C++...(公共分类列表)" << endl;
}
void content()
{
    cout << "C++学科视频" << endl;
}
};

void test01()
{
    //Java页面
    cout << "Java下载视频页面如下：" << endl;
    Java ja;
    ja.header();
    ja.footer();
    ja.left();
    ja.content();
    cout << "-----" << endl;

    //Python页面
    cout << "Python下载视频页面如下：" << endl;
    Python py;
    py.header();
    py.footer();
    py.left();
    py.content();
    cout << "-----" << endl;

    //C++页面
    cout << "C++下载视频页面如下：" << endl;
    CPP cpp;
    cpp.header();
    cpp.footer();
    cpp.left();
    cpp.content();
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

继承实现：

```
//公共页面
class BasePage
{
public:
    void header()
    {
        cout << "首页、公开课、登录、注册... (公共头部)" << endl;
    }

    void footer()
    {
        cout << "帮助中心、交流合作、站内地图... (公共底部)" << endl;
    }
    void left()
    {
        cout << "Java,Python,C++... (公共分类列表)" << endl;
    }

};

//Java页面
class Java : public BasePage           //继承的结构
{
public:
    void content()
    {
        cout << "JAVA学科视频" << endl;
    }

};

//Python页面
class Python : public BasePage
{
public:
    void content()
    {
        cout << "Python学科视频" << endl;
    }

};

//C++页面
class CPP : public BasePage
{
public:
    void content()
    {
        cout << "C++学科视频" << endl;
    }

};

void test01()
{
    //Java页面
    cout << "Java下载视频页面如下：" << endl;
    Java ja;
    ja.header();
    ja.footer();
```

```

        ja.left();
        ja.content();
        cout << "-----" << endl;

    //Python页面
    cout << "Python下载视频页面如下：" << endl;
    Python py;
    py.header();
    py.footer();
    py.left();
    py.content();
    cout << "-----" << endl;

    //C++页面
    cout << "C++下载视频页面如下：" << endl;
    CPP cp;
    cp.header();
    cp.footer();
    cp.left();
    cp.content();

}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：

继承的好处：可以减少重复的代码

class A : public B; //继承的语法

A 类称为子类 或 派生类

B 类称为父类 或 基类

派生类中的成员，包含两大部分：

一类是从基类继承过来的，一类是自己增加的成员。

从基类继承过来的表现其共性，而新增的成员体现了其个性。

4.6.2 继承方式

继承的语法： class 子类 : 继承方式 父类

继承方式一共有三种：

- 公共继承
- 保护继承
- 私有继承



示例：

```
class Base1
{
public:
    int m_A;
protected:
    int m_B;
private:
    int m_C;
};

//公共继承
class Son1 :public Base1
{
public:
    void func()
    {
        m_A; //可访问 public权限
        m_B; //可访问 protected权限
        //m_C; //不可访问
    }
};

void myClass()
{
    Son1 s1;
    s1.m_A; //其他类只能访问到公共权限
}

//保护继承
class Base2
{
public:
    int m_A;
protected:
    int m_B;
private:
    int m_C;
};

class Son2:protected Base2
{
public:
    void func()
    {
        m_A; //可访问 protected权限
        m_B; //可访问 protected权限
        //m_C; //不可访问
    }
};

void myClass2()
{
    Son2 s;
    //s.m_A; //不可访问
}

//私有继承
```

```
class Base3
{
public:
    int m_A;
protected:
    int m_B;
private:
    int m_C;
};

class Son3:private Base3
{
public:
    void func()
    {
        m_A; //可访问 private权限
        m_B; //可访问 private权限
        //m_C; //不可访问
    }
};

class GrandSon3 :public Son3
{
public:
    void func()
    {
        //Son3是私有继承，所以继承Son3的属性在GrandSon3中都无法访问到
        //m_A;
        //m_B;
        //m_C;
    }
};
```

4.6.3 继承中的对象模型

问题：从父类继承过来的成员，哪些属于子类对象中？

示例：

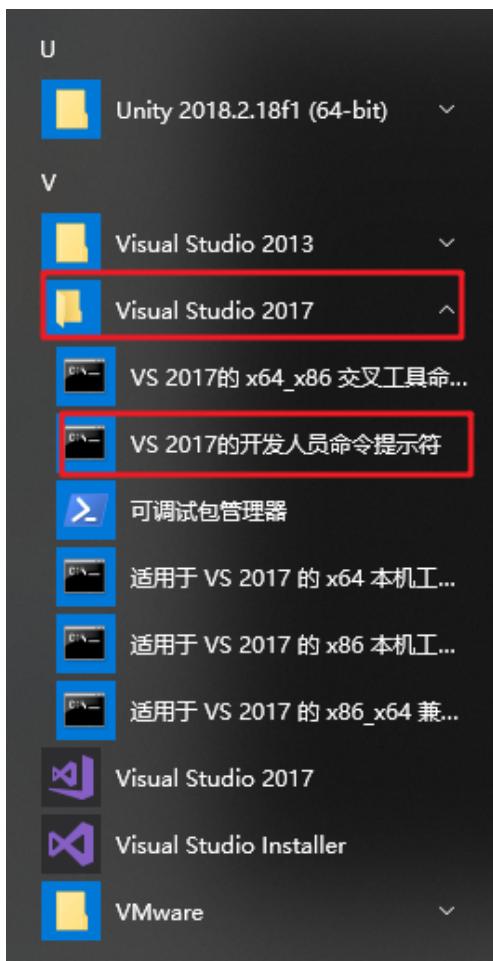
```
class Base
{
public:
    int m_A;
protected:
    int m_B;
private:
    int m_C; //私有成员只是被隐藏了，但是还是会继承下去
};

//公共继承
class Son :public Base
{
public:
    int m_D;
};

void test01()
{
    cout << "sizeof Son = " << sizeof(Son) << endl; //4个int类型=16
}

int main() {
    test01();
    system("pause");
    return 0;
}
```

利用工具查看：



打开工具窗口后，定位到当前CPP文件的盘符

然后输入： cl /d1 reportSingleClassLayout查看的类名 所属文件名

效果如下图：

```
*****
C:\Program Files (x86)\Microsoft Visual Studio\2017\Community>F:
F:\>cd F:\VS项目\继承\继承
F:\VS项目\继承\继承>cl /d1 reportSingleClassLayoutSon "03 继承中的对象模型.cpp"
用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.15.26732.1 版
版权所有 (C) Microsoft Corporation。保留所有权利。
03 继承中的对象模型.cpp
C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.15.26726\include\xlocale(319): warning C4
530: 使用了 C++ 异常处理程序，但未启用展开语义。请指定 /EHsc

class Son      size(16):
    +--+
    0     |--- (base class Base)
    0     |   m_A
    4     |   m_B
    8     |   m_C
    12    |--- m_D
        +--+
父类继承过来的
子类特有的

Microsoft (R) Incremental Linker Version 14.15.26732.1
Copyright (C) Microsoft Corporation. All rights reserved.
"/out:03 继承中的对象模型.exe"
"03 继承中的对象模型.obj"
F:\VS项目\继承\继承>
```

结论：父类中私有成员也是被子类继承下去了，只是由编译器给隐藏后访问不到

4.6.4 继承中构造和析构顺序

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

问题：父类和子类的构造和析构顺序是谁先谁后？

示例：

```
class Base
{
public:
    Base()
    {
        cout << "Base构造函数!" << endl;
    }
    ~Base()
    {
        cout << "Base析构函数!" << endl;
    }
};

class Son : public Base
{
public:
    Son()
    {
        cout << "Son构造函数!" << endl;
    }
    ~Son()
    {
        cout << "Son析构函数!" << endl;
    }
};

void test01()
{
    //继承中 先调用父类构造函数（得现有父类才能进行继承），再调用子类构造函数，析构顺序与构造相反，（B类先析构）
    Son s;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：继承中 先调用父类构造函数，再调用子类构造函数，析构顺序与构造相反

4.6.5 继承同名成员处理方式

问题：当子类与父类出现同名的成员，如何通过子类对象，访问到子类或父类中同名的数据呢？

- 访问子类同名成员 直接访问即可
- 访问父类同名成员 需要加作用域

示例：

```
class Base {
public:
    Base()
    {
        m_A = 100;
    }

    void func()
    {
        cout << "Base - func()调用" << endl;
    }

    void func(int a)
    {
        cout << "Base - func(int a)调用" << endl;
    }

public:
    int m_A;
};

class Son : public Base {
public:
    Son()
    {
        m_A = 200;
    }

    //当子类与父类拥有同名的成员函数，子类会隐藏父类中所有版本的同名成员函数（包括重载版本的）
    //如果想访问父类中被隐藏的同名成员函数，需要加父类的作用域（同名变量同理）
    void func()
    {
        cout << "Son - func()调用" << endl;
    }

public:
    int m_A;
};

void test01()
{
    Son s;

    cout << "Son下的m_A = " << s.m_A << endl;
    cout << "Base下的m_A = " << s.Base::m_A << endl; //加作用域

    s.func();
    s.Base::func();
    s.Base::func(10);

}
int main() {

    test01();
```

```
    system("pause");
    return EXIT_SUCCESS;
}
```

总结：

1. 子类对象可以直接访问到子类中同名成员
2. 子类对象加作用域可以访问到父类同名成员
3. 当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

4.6.6 继承同名静态成员处理方式

问题：继承中同名的静态成员在子类对象上如何进行访问？

静态成员和非静态成员出现同名，处理方式一致

- 访问子类同名成员 直接访问即可
- 访问父类同名成员 需要加作用域

示例：

```
class Base {
public:
    static void func()
    {
        cout << "Base - static void func()" << endl;
    }
    static void func(int a)
    {
        cout << "Base - static void func(int a)" << endl;
    }

    static int m_A;
};

int Base::m_A = 100;

class Son : public Base {
public:
    static void func()
    {
        cout << "Son - static void func()" << endl;
    }
    static int m_A;
};

int Son::m_A = 200;

//同名成员属性
void test01()
{
    //通过对象访问
    cout << "通过对象访问: " << endl;
    Son s;
    cout << "Son 下 m_A = " << s.m_A << endl;
    cout << "Base 下 m_A = " << s.Base::m_A << endl;

    //通过类名访问
    cout << "通过类名访问: " << endl;
    cout << "Son 下 m_A = " << Son::m_A << endl;
    cout << "Base 下 m_A = " << Son::Base::m_A << endl;
}

//同名成员函数
void test02()
{
    //通过对象访问
    cout << "通过对象访问: " << endl;
    Son s;
    s.func();
    s.Base::func();

    cout << "通过类名访问: " << endl;
    Son::func();
    Son::Base::func();
    //出现同名，子类会隐藏掉父类中所有同名成员函数（包括重载成员函数），需要加作作用域访问
```

```
Son::Base::func(100);
}
int main() {
    //test01();
    test02();

    system("pause");

    return 0;
}
```

| 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）

4.6.7 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1 , 继承方式 父类2...

多继承可能会引发不同父类中有同名成员出现，需要加作用域区分

C++实际开发中不建议用多继承

示例：

```

class Base1 {
public:
    Base1()
    {
        m_A = 100;
    }
public:
    int m_A;
};

class Base2 {
public:
    Base2()
    {
        m_A = 200; //开始是m_B 不会出问题，但是改为mA就会出现不明确
    }
public:
    int m_A;
};

//语法: class 子类: 继承方式 父类1 , 继承方式 父类2
class Son : public Base2, public Base1
{
public:
    Son()
    {
        m_C = 300;
        m_D = 400;
    }
public:
    int m_C;
    int m_D;
};

//多继承容易产生成员同名的情况
//通过使用类名作用域可以区分调用哪一个基类的成员
void test01()
{
    Son s;
    cout << "sizeof Son = " << sizeof(s) << endl;
    cout << s.Base1::m_A << endl;
    cout << s.Base2::m_A << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：多继承中如果父类中出现了同名情况，子类使用时候要加作用域

4.6.8 菱形继承

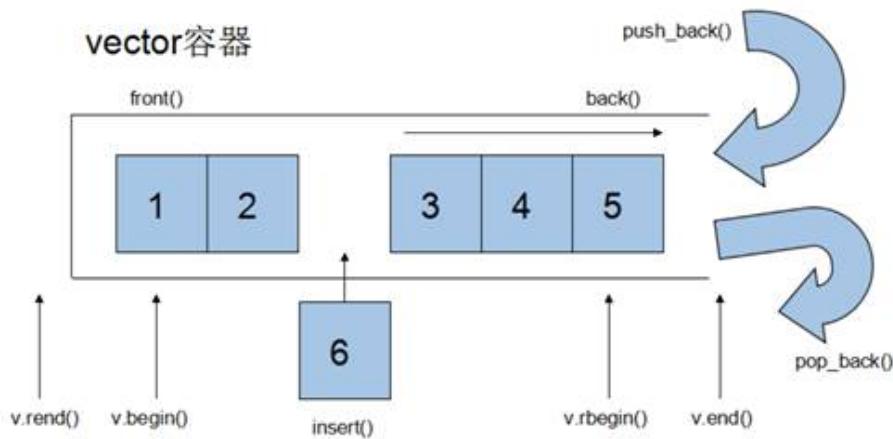
菱形继承概念：

两个派生类继承同一个基类

又有某个类同时继承者两个派生类

这种继承被称为菱形继承，或者钻石继承

典型的菱形继承案例：



菱形继承问题：

1. 羊继承了动物的数据，驼同样继承了动物的数据，当草泥马使用数据时，就会产生二义性。
2. 草泥马继承自动物的数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以。

示例：

```

class Animal
{
public:
    int m_Age;
};

//继承前加virtual关键字后，变为虚继承
//此时公共的父类Animal称为虚基类
class Sheep : virtual public Animal {};
class Tuo : virtual public Animal {};
class SheepTuo : public Sheep, public Tuo {};

void test01()
{
    SheepTuo st;
    st.Sheep::m_Age = 100;
    st.Tuo::m_Age = 200;
    // 当菱形继承，两个父类拥有相同的数据，需要加以作用域区分
    cout << "st.Sheep::m_Age = " << st.Sheep::m_Age << endl; //200
    cout << "st.Tuo::m_Age = " << st.Tuo::m_Age << endl; //200
    cout << "st.m_Age = " << st.m_Age << endl; //200
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：

- 菱形继承带来的主要问题是子类继承两份相同的数据，导致资源浪费以及毫无意义
- 利用虚继承可以解决菱形继承问题，虚继承会只保留一个同名数据

4.7 多态

4.7.1 多态的基本概念

多态是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多态

静态多态和动态多态区别：

- 静态多态的函数地址早绑定 - 编译阶段确定函数地址
- 动态多态的函数地址晚绑定 - 运行阶段确定函数地址

下面通过案例进行讲解多态

```
class Animal
{
public:
    //Speak函数就是虚函数
    //函数前面加上virtual关键字，变成虚函数，那么编译器在编译的时候就不能确定函数调用了。（地址晚绑定）
    virtual void speak()
    {
        cout << "动物在说话" << endl;
    }
};

class Cat :public Animal
{
public:
    void speak()
    {
        cout << "小猫在说话" << endl;
    }
};

class Dog :public Animal
{
public:
    void speak()
    {
        cout << "小狗在说话" << endl;
    }
};

};

//我们希望传入什么对象，那么就调用什么对象的函数
//如果函数地址在编译阶段就能确定，那么静态联编
//如果函数地址在运行阶段才能确定，就是动态联编

void DoSpeak(Animal & animal)
{
    animal.speak();
}

//多态满足条件：
//1、有继承关系
//2、子类重写父类中的虚函数（函数名 返回值 参数值完全一致是重写）
//多态使用：
//父类指针或引用指向子类对象

void test01()
{
    Cat cat;
    DoSpeak(cat);           //ful

    Dog dog;
    DoSpeak(dog);
}
```

```

int main() {
    test01();
    system("pause");
    return 0;
}

```

vptr -虚函数（表）指针

vftable 虚函数表里记录虚函数的地址 比如: &Animal::speak

```

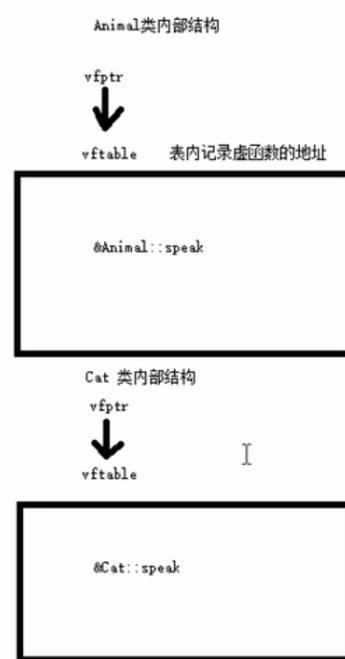
class Animal
{
public:
    //虚函数
    virtual void speak()
    {
        cout << "动物在说话" << endl;
    }
};

//猫类
class Cat : public Animal
{
public:
    //重写 函数返回值类型 函数名 参数列表 完全相同
    virtual void speak()
    {
        cout << "小猫在说话" << endl;
    }
};

当子类重写父类的虚函数

子类中的虚函数表 内部 会替换成 子类的虚函数地址

```



vptr - 虚函数（表）指针
v - virtual
f - function
ptr - pointer

vftable - 虚函数表
v - virtual
f - function
table - table

当父类的指针或者引用指向子类对象时候，发生多态
Animal & animal = cat;
animal.speak();

总结：

多态满足条件

- 有继承关系
- 子类重写父类中的虚函数

多态使用条件

- 父类指针或引用指向子类对象

重写：函数返回值类型 函数名 参数列表 完全一致称为重写

4.7.2 多态案例一-计算器类

案例描述：

分别利用普通写法和多态技术，设计实现两个操作数进行运算的计算器类

多态的优点：

- 代码组织结构清晰
- 可读性强
- 利于前期和后期的扩展以及维护

示例：

```

//普通实现
class Calculator {
public:
    int getResult(string oper)
    {
        if (oper == "+") {
            return m_Num1 + m_Num2;
        }
        else if (oper == "-") {
            return m_Num1 - m_Num2;
        }
        else if (oper == "*") {
            return m_Num1 * m_Num2;
        }
        //如果要提供新的运算，需要修改源码
        //在真实开发中 提倡开闭原则
        //开闭原则：对扩展进行开发，对修改进行关闭
    }
public:
    int m_Num1;
    int m_Num2;
};

void test01()
{
    //普通实现测试
    Calculator c;
    c.m_Num1 = 10;
    c.m_Num2 = 10;
    cout << c.m_Num1 << " + " << c.m_Num2 << " = " << c.getResult["+") << endl;

    cout << c.m_Num1 << " - " << c.m_Num2 << " = " << c.getResult["-") << endl;

    cout << c.m_Num1 << " * " << c.m_Num2 << " = " << c.getResult["*") << endl;
}

```

```

//多态实现
//抽象计算器类
//多态优点：代码组织结构清晰，可读性强，利于前期和后期的扩展以及维护
class AbstractCalculator
{
public :

    virtual int getResult()
    {
        return 0;
    }

    int m_Num1;
    int m_Num2;
};

//加法计算器

```

```
class AddCalculator :public AbstractCalculator
{
public:
    int getResult()
    {
        return m_Num1 + m_Num2;
    }
};

//减法计算器
class SubCalculator :public AbstractCalculator
{
public:
    int getResult()
    {
        return m_Num1 - m_Num2;
    }
};

//乘法计算器
class MulCalculator :public AbstractCalculator
{
public:
    int getResult()
    {
        return m_Num1 * m_Num2;
    }
};

void test02()
{
    //创建加法计算器
    //父类指针或引用指向子类对象
    AbstractCalculator *abc = new AddCalculator;
    abc->m_Num1 = 10;
    abc->m_Num2 = 10;
    cout << abc->m_Num1 << " + " << abc->m_Num2 << " = " << abc->getResult() << endl;
    delete abc; //用完了记得销毁

    //创建减法计算器
    abc = new SubCalculator;
    abc->m_Num1 = 10;
    abc->m_Num2 = 10;
    cout << abc->m_Num1 << " - " << abc->m_Num2 << " = " << abc->getResult() << endl;
    delete abc;

    //创建乘法计算器
    abc = new MulCalculator;
    abc->m_Num1 = 10;
    abc->m_Num2 = 10;
    cout << abc->m_Num1 << " * " << abc->m_Num2 << " = " << abc->getResult() << endl;
    delete abc;
}
```

```
int main() {  
    //test01();  
  
    test02();  
  
    system("pause");  
  
    return 0;  
}
```

| 总结：C++开发提倡利用多态设计程序架构，因为多态优点很多

4.7.3 纯虚函数和抽象类

在多态中，通常父类中虚函数的实现是毫无意义的，主要都是调用子类重写的内容

因此可以将虚函数改为**纯虚函数**

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

示例：

```

class Base
{
public:
    //纯虚函数
    //类中只要有一个纯虚函数就称为抽象类
    //抽象类无法实例化对象
    //子类必须重写父类中的纯虚函数，否则也属于抽象类
    virtual void func() = 0;
};

class Son :public Base
{
public:
    virtual void func()
    {
        cout << "func调用" << endl;
    };
};

void test01()
{
    Base * base = NULL;
    //base = new Base; // 错误，抽象类无法实例化对象
    base = new Son;
    base->func();
    delete base;//记得销毁
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

4.7.4 多态案例二-制作饮品

案例描述：

制作饮品的大致流程为：煮水 - 冲泡 - 倒入杯中 - 加入辅料

利用多态技术实现本案例，提供抽象制作饮品基类，提供子类制作咖啡和茶叶

1、煮水

2、冲泡咖啡

3、倒入杯中

4、加糖和牛奶



冲咖啡

1、煮水

2、冲泡茶叶

3、倒入杯中

4、加柠檬



冲茶叶

示例：

```
//抽象制作饮品
class AbstractDrinking {
public:
    //烧水
    virtual void Boil() = 0;
    //冲泡
    virtual void Brew() = 0;
    //倒入杯中
    virtual void PourInCup() = 0;
    //加入辅料
    virtual void PutSomething() = 0;
    //规定流程
    void MakeDrink() {
        Boil();
        Brew();
        PourInCup();
        PutSomething();
    }
};

//制作咖啡
class Coffee : public AbstractDrinking {
public:
    //烧水
    virtual void Boil() {
        cout << "煮农夫山泉!" << endl;
    }
    //冲泡
    virtual void Brew() {
        cout << "冲泡咖啡!" << endl;
    }
    //倒入杯中
    virtual void PourInCup() {
        cout << "将咖啡倒入杯中!" << endl;
    }
    //加入辅料
    virtual void PutSomething() {
        cout << "加入牛奶!" << endl;
    }
};

//制作茶水
class Tea : public AbstractDrinking {
public:
    //烧水
    virtual void Boil() {
        cout << "煮自来水!" << endl;
    }
    //冲泡
    virtual void Brew() {
        cout << "冲泡茶叶!" << endl;
    }
    //倒入杯中
    virtual void PourInCup() {
        cout << "将茶水倒入杯中!" << endl;
    }
};
```

```

    }
    //加入辅料
    virtual void PutSomething() {
        cout << "加入枸杞!" << endl;
    }
};

//业务函数
void DoWork(AbstractDrinking* drink) {
// 这里试了一下，用引用也可以 就不用delete删除了堆区数据了，下面传入的改成子类的实例化对象
    drink->MakeDrink();
    delete drink;
}

void test01() {
    DoWork(new Coffee);
    cout << "-----" << endl;
    DoWork(new Tea);
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

4.7.5 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：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(){} (类外写纯虚析构的实现)

示例：

```

class Animal {
public:

    Animal()
    {
        cout << "Animal 构造函数调用! " << endl;
    }
    virtual void Speak() = 0;

    //析构函数加上virtual关键字，变成虚析构函数
    //virtual ~Animal()
    //{
    //    cout << "Animal虚析构函数调用! " << endl;
    //}

    virtual ~Animal() = 0;
};

Animal::~Animal()           //虚析构和纯虚析构都需要代码实现 不然无法释放内存
{
    cout << "Animal 纯虚析构函数调用! " << endl;
}

//和包含普通纯虚函数的类一样，包含了纯虚析构函数的类也是一个抽象类。不能够被实例化。

class Cat : public Animal {
public:
    Cat(string name)
    {
        cout << "Cat构造函数调用! " << endl;
        m_Name = new string(name);
    }
    virtual void Speak()
    {
        cout << *m_Name << "小猫在说话!" << endl;
    }
    ~Cat()
    {
        cout << "Cat析构函数调用!" << endl;
        if (this->m_Name != NULL) {
            delete m_Name;
            m_Name = NULL;
        }
    }

public:
    string *m_Name;
};

void test01()
{
    Animal *animal = new Cat("Tom");
    animal->Speak();
}

```

```

//通过父类指针去释放，会导致子类对象可能清理不干净，造成内存泄漏
//怎么解决？给基类增加一个虚析构函数
//虚析构函数就是用来解决通过父类指针释放子类对象
delete animal;
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：

1. 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象
2. 如果子类中没有堆区数据，可以不写为虚析构或纯虚析构
3. 拥有纯虚析构函数的类也属于抽象类

4.7.6 多态案例三-电脑组装

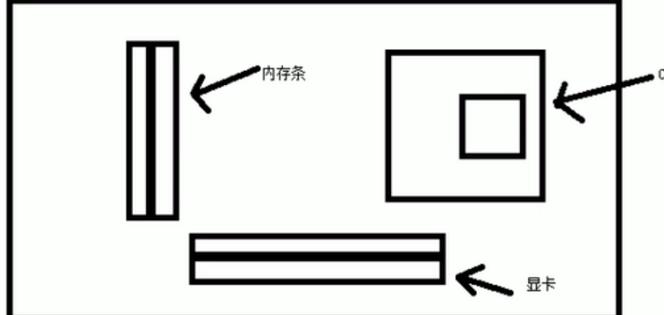
案例描述：

电脑主要组成部件为 CPU（用于计算）, 显卡（用于显示）, 内存条（用于存储）

将每个零件封装出抽象基类，并且提供不同的厂商生产不同的零件，例如Intel厂商和Lenovo厂商

创建电脑类提供让电脑工作的函数，并且调用每个零件工作的接口

测试时组装三台不同的电脑进行工作



```

抽象出每个零件的类
class CPU 抽象类
{
    //抽象计算函数
    virtual void calculate() = 0;
}

class VideoCard 抽象类
{
    //抽象显示函数
    virtual void display() = 0;
}

class Memory 抽象类
{
    //抽象存储函数
    virtual void storage() = 0;
}

电脑类
class Computer
{
    构造函数中传入三个零件指针
    提供工作的函数
    {
        调用每个零件工作的接口
    }
}

具体零件厂商
Intel 厂商
class IntelCpu : public CPU
{
    void calculate()
    {
        cout << "Intel的CPU开始计算了！"
    }
}

Lenovo 厂商
也需要提供三个零件

```

测试阶段 组装三台不同的电脑

示例：

```
#include<iostream>
using namespace std;

//抽象CPU类
class CPU
{
public:
    //抽象的计算函数
    virtual void calculate() = 0;
};

//抽象显卡类
class VideoCard
{
public:
    //抽象的显示函数
    virtual void display() = 0;
};

//抽象内存条类
class Memory
{
public:
    //抽象的存储函数
    virtual void storage() = 0;
};

//电脑类
class Computer
{
public:
    Computer(CPU * cpu, VideoCard * vc, Memory * mem)
    {
        m_cpu = cpu;
        m_vc = vc;
        m_mem = mem;
    }

    //提供工作的函数
    void work()
    {
        //让零件工作起来，调用接口
        m_cpu->calculate();

        m_vc->display();

        m_mem->storage();
    }

    //提供析构函数 释放3个电脑零件
    ~Computer()
    {

        //释放CPU零件
        if (m_cpu != NULL)
```

```

    {
        delete m_cpu;
        m_cpu = NULL;
    }

    //释放显卡零件
    if (m_vc != NULL)
    {
        delete m_vc;
        m_vc = NULL;
    }

    //释放内存条零件
    if (m_mem != NULL)
    {
        delete m_mem;
        m_mem = NULL;
    }
}

private:

CPU * m_cpu; //CPU的零件指针
VideoCard * m_vc; //显卡零件指针
Memory * m_mem; //内存条零件指针
};

//具体厂商
//Intel厂商
class IntelCPU :public CPU
{
public:
    virtual void calculate()
    {
        cout << "Intel的CPU开始计算了! " << endl;
    }
};

class IntelVideoCard :public VideoCard
{
public:
    virtual void display()
    {
        cout << "Intel的显卡开始显示了! " << endl;
    }
};

class IntelMemory :public Memory
{
public:
    virtual void storage()
    {
        cout << "Intel的内存条开始存储了! " << endl;
    }
};

```

```
//Lenovo厂商
class LenovoCPU :public CPU
{
public:
    virtual void calculate()
    {
        cout << "Lenovo的CPU开始计算了！" << endl;
    }
};

class LenovoVideoCard :public VideoCard
{
public:
    virtual void display()
    {
        cout << "Lenovo的显卡开始显示了！" << endl;
    }
};

class LenovoMemory :public Memory
{
public:
    virtual void storage()
    {
        cout << "Lenovo的内存条开始存储了！" << endl;
    }
};

void test01()
{
    //第一台电脑零件
    CPU * intelCpu = new IntelCPU;
    VideoCard * intelCard = new IntelVideoCard;
    Memory * intelMem = new IntelMemory;

    cout << "第一台电脑开始工作：" << endl;
    //创建第一台电脑
    Computer * computer1 = new Computer(intelCpu, intelCard, intelMem);
    computer1->work();
    delete computer1;

    cout << "-----" << endl;
    cout << "第二台电脑开始工作：" << endl;
    //第二台电脑组装
    Computer * computer2 = new Computer(new LenovoCPU, new LenovoVideoCard, new LenovoMemory);
    computer2->work();
    delete computer2;

    cout << "-----" << endl;
    cout << "第三台电脑开始工作：" << endl;
    //第三台电脑组装
    Computer * computer3 = new Computer(new LenovoCPU, new IntelVideoCard, new LenovoMemory);
    computer3->work();
}
```

```
    delete computer3;  
}
```

5 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 < fstream >

文件类型分为两种：

1. **文本文件** - 文件以文本的**ASCII码**形式存储在计算机中
2. **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它们

操作文件的三大类：

1. ofstream：写操作 (output file stream)
2. ifstream：读操作
3. fstream：读写操作

5.1 文本文件

5.1.1 写文件

写文件步骤如下：

1. 包含头文件
`#include <fstream>`
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 << "写入的数据";`
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：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

注意： 文件打开方式可以配合使用，利用|操作符

例如：用二进制方式写文件 `ios::binary | ios::out`

示例：

```
#include <fstream>

void test01()
{
    ofstream ofs; //也可以用fstream类
    ofs.open("/home/donghao/Documents/project/study/test.txt", ios::out);
//使用vscode时最好写绝对路径，只写文件名会被写到gcc编译器所在文件夹里
    ofs << "姓名：张三" << endl;
    ofs << "性别：男" << endl;
    ofs << "年龄：18" << endl;

    ofs.close();
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结：

- 文件操作必须包含头文件 `fstream`
- 读文件可以利用 `ofstream`，或者 `fstream` 类
- 打开文件时候需要指定操作文件的路径，以及打开方式
- 利用 `<<` 可以向文件中写数据
- 操作完毕，要关闭文件

5.1.2 读文件

读文件与写文件步骤相似，但是读取方式相对于比较多

读文件步骤如下：

1. 包含头文件

```
#include <fstream>
```

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

四种方式读取

5. 关闭文件

```
ifs.close();
```

示例：

```
#include <fstream>
#include <string>
void test01()
{
    ifstream ifs;
    ifs.open("/home/donghao/Documents/project/study/test.txt", ios::in);

    if (!ifs.is_open())
    {
        cout << "文件打开失败" << endl;
        return;
    }

    // //第一种方式
    // char buf[1024] = { 0 };
    // while (ifs >> buf)
    // {
    //     cout << buf << endl;
    // }

    // //第二种
    // char buf[1024] = { 0 };
    // while (ifs.getline(buf, sizeof(buf)))
    // {
    //     cout << buf << endl;
    // }

    // //第三种
    // string buf;
    // while (getline(ifs, buf))
    // {
    //     cout << buf << endl;
    // }

    char c; //一个一个字符读取 效率太慢 不太实用
    while ((c = ifs.get()) != EOF) //EOF end of file 文件尾
    {
        cout << c;
    }

    ifs.close();
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结：

- 读文件可以利用 ifstream，或者fstream类
- 利用is_open函数可以判断文件是否打开成功
- close 关闭文件

5.2 二进制文件

以二进制的方式对文件进行读写操作

打开方式要指定为 ios::binary

5.2.1 写文件

二进制方式写文件主要利用流对象调用成员函数write

函数原型： ostream& write(const char * buffer,int len);

参数解释：字符指针buffer指向内存中一段存储空间。len是读写的字节数

示例：

```

#include <iostream>
#include <string>

class Person
{
public:
    char m_Name[64];
    int m_Age;
};

//二进制文件 写文件
void test01()
{
    //1、包含头文件

    //2、创建输出流对象
    ofstream ofs("person.txt", ios::out | ios::binary); //有构造函数

    //3、打开文件
    //ofs.open("person.txt", ios::out | ios::binary);

    Person p = {"张三" , 18};

    //4、写文件
    ofs.write((const char *)&p, sizeof(p)); //const char * 强转型

    //5、关闭文件
    ofs.close();
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 文件输出流对象 可以通过write函数，以二进制方式写数据

5.2.2 读文件

二进制方式读文件主要利用流对象调用成员函数read

函数原型： istream& read(char *buffer, int len);

参数解释：字符指针buffer指向内存中一段存储空间。len是读写的字节数

示例：

```

#include <iostream>
#include <string>

class Person
{
public:
    char m_Name[64];
    int m_Age;
};

void test01()
{
    ifstream ifs("person.txt", ios::in | ios::binary);
    if (!ifs.is_open())
    {
        cout << "文件打开失败" << endl;
    }

    Person p;
    ifs.read((char *)&p, sizeof(p));

    cout << "姓名: " << p.m_Name << " 年龄: " << p.m_Age << endl;
    ifs.close();
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

- 文件输入流对象 可以通过read函数，以二进制方式读数据

C++提高编程

- 本阶段主要针对C++泛型编程和STL技术做详细讲解，探讨C++更深层的使用

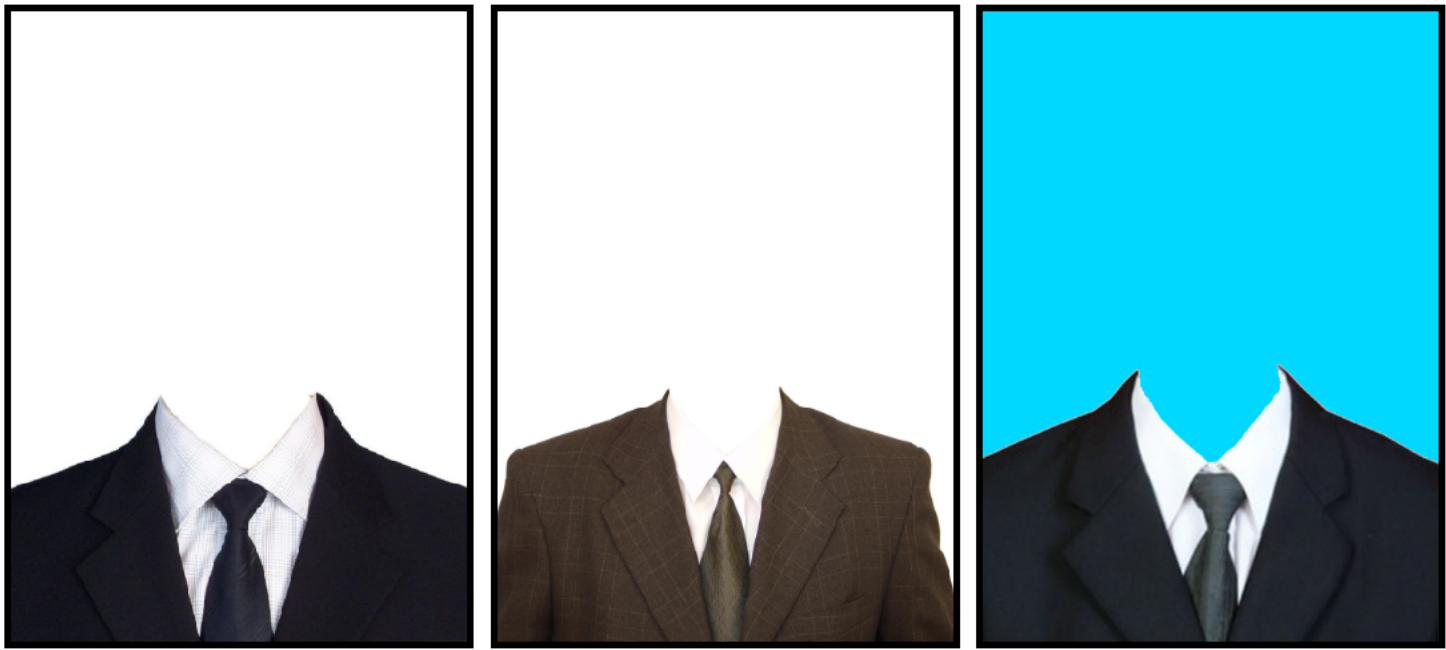
1 模板

1.1 模板的概念

模板就是建立**通用的模具**，大大**提高复用性**

例如生活中的模板

一寸照片模板：



PPT模板：

A collection of PPT slide designs for a business report. The top row shows two full slide presentations on a desktop monitor. The left monitor displays a dark purple slide with a circular graphic and the text '201X 年终总结报告'. The right monitor displays a light blue slide with a geometric pattern and the text '201X BUSINESS REPORT 年终工作总结模板'. Below these are four smaller preview slides: 1. A 'CONTENTS' slide with a globe graphic and a grid of icons. 2. A slide with a pink circular graphic and the text 'H 201X 年终总结'. 3. A slide with a blue background and a grid of icons, listing '01 年度工作概述', '02 工作进展回顾', '03 成功经验分享', and '04 未来工作计划'. 4. Two '01 年度工作概述' slides, each featuring a grid of icons and text sections.



模板的特点：

- 模板不可以直接使用，它只是一个框架
- 模板的通用并不是万能的

1.2 函数模板

- C++另一种编程思想称为 泛型编程，主要利用的技术就是模板
- C++提供两种模板机制:**函数模板**和**类模板**

1.2.1 函数模板语法

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：

```
template<typename T>
```

函数声明或定义

解释：

template --- 声明创建模板

typename --- 表面其后面的符号是一种数据类型，可以用class代替

T --- 通用的数据类型，名称可以替换，通常为大写字母

示例：

```
//交换整型函数
void swapInt(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

//交换浮点型函数
void swapDouble(double& a, double& b) {
    double temp = a;
    a = b;
    b = temp;
}

//利用模板提供通用的交换函数
template<typename T>
void mySwap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

void test01()
{
    int a = 10;
    int b = 20;

    //swapInt(a, b);

    //利用模板实现交换
    //1、自动类型推导
    mySwap(a, b);

    //2、显示指定类型
    mySwap<int>(a, b);

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结：

- 函数模板利用关键字 template
- 使用函数模板有两种方式：自动类型推导、显示指定类型
- 模板的目的是为了将类型参数化，提高复用性

1.2.2 函数模板注意事项

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

示例：

```

//利用模板提供通用的交换函数
template<class T> //typename可以替换为class，效果一样
void mySwap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

// 1、自动类型推导，必须推导出一致的数据类型T，才可以使用
void test01()
{
    int a = 10;
    int b = 20;
    char c = 'c';

    mySwap(a, b); // 正确，可以推导出一致的T
    //mySwap(a, c); // 错误，推导不出一致的T类型
}

// 2、模板必须要确定出T的数据类型，才可以使用
template<class T>
void func()
{
    cout << "func 调用" << endl;
}

void test02()
{
    //func(); //错误，模板不能独立使用，必须确定出T的类型
    func<int>(); //利用显示指定类型的方式，给T一个类型，才可以使用该模板
}

int main() {
    test01();
    test02();

    system("pause");
    return 0;
}

```

总结：

- 使用模板时必须确定出通用数据类型T，并且能够推导出一致的类型

1.2.3 函数模板案例

案例描述：

- 利用函数模板封装一个排序的函数，可以对**不同数据类型数组**进行排序
- 排序规则从大到小，排序算法为**选择排序**
- 分别利用**char数组**和**int数组**进行测试

示例：

```
//交换的函数模板
template<typename T>
void mySwap(T &a, T&b)
{
    T temp = a;
    a = b;
    b = temp;
}

template<class T> // 也可以替换成typename
//利用选择排序，进行对数组从大到小的排序
void mySort(T arr[], int len)
{
    for (int i = 0; i < len; i++)
    {
        int max = i; //最大数的下标
        for (int j = i + 1; j < len; j++)
        {
            if (arr[max] < arr[j])
            {
                max = j;
            }
        }
        if (max != i) //如果最大数的下标不是i，交换两者
        {
            mySwap(arr[max], arr[i]);
        }
    }
}
template<typename T>
void printArray(T arr[], int len) {

    for (int i = 0; i < len; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
void test01()
{
    //测试char数组
    char charArr[] = "bdcfeagh";
    int num = sizeof(charArr) / sizeof(char);
    mySort(charArr, num);
    printArray(charArr, num);
}

void test02()
{
    //测试int数组
    int intArr[] = { 7, 5, 8, 1, 3, 9, 2, 4, 6 };
    int num = sizeof(intArr) / sizeof(int);
    mySort(intArr, num);
    printArray(intArr, num);
}
```

```
int main() {  
    test01();  
    test02();  
  
    system("pause");  
  
    return 0;  
}
```

总结：模板可以提高代码复用，需要熟练掌握

1.2.4 普通函数与函数模板的区别

普通函数与函数模板区别：

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

示例：

```

//普通函数
int myAdd01(int a, int b)
{
    return a + b;
}

//函数模板
template<class T>
T myAdd02(T a, T b)
{
    return a + b;
}

//使用函数模板时，如果用自动类型推导，不会发生自动类型转换，即隐式类型转换
void test01()
{
    int a = 10;
    int b = 20;
    char c = 'c';

    cout << myAdd01(a, c) << endl; //正确，将char类型的'c'隐式转换为int类型 'c' 对应 ASCII码 99

    //myAdd02(a, c); // 报错，使用自动类型推导时，不会发生隐式类型转换

    myAdd02<int>(a, c); //正确，如果用显示指定类型，可以发生隐式类型转换
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：建议使用显示指定类型的方式，调用函数模板，因为可以自己确定通用类型T

1.2.5 普通函数与函数模板的调用规则

调用规则如下：

1. 如果函数模板和普通函数都可以实现，优先调用普通函数
2. 可以通过空模板参数列表来强制调用函数模板
3. 函数模板也可以发生重载
4. 如果函数模板可以产生更好的匹配，优先调用函数模板

示例：

```

//普通函数与函数模板调用规则
void myPrint(int a, int b)
{
    cout << "调用的普通函数" << endl;
}

template<typename T>
void myPrint(T a, T b)
{
    cout << "调用的模板" << endl;
}

template<typename T>
void myPrint(T a, T b, T c)
{
    cout << "调用重载的模板" << endl;
}

void test01()
{
    //1、如果函数模板和普通函数都可以实现，优先调用普通函数
    // 注意 如果告诉编译器 普通函数是有的，但只是声明没有实现，或者不在当前文件内实现，就会报错找不到
    int a = 10;
    int b = 20;
    myPrint(a, b); //调用普通函数

    //2、可以通过空模板参数列表来强制调用函数模板
    myPrint<>(a, b); //调用函数模板

    //3、函数模板也可以发生重载
    int c = 30;
    myPrint(a, b, c); //调用重载的函数模板

    //4、如果函数模板可以产生更好的匹配，优先调用函数模板
    char c1 = 'a';
    char c2 = 'b';
    myPrint(c1, c2); //调用函数模板
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：既然提供了函数模板，最好就不要提供普通函数，否则容易出现二义性

1.2.6 模板的局限性

局限性：

- 模板的通用性并不是万能的

例如：

```
template<class T>
void f(T a, T b)
{
    a = b;
}
```

在上述代码中提供的赋值操作，如果传入的a和b是一个数组，就无法实现了

再例如：

```
template<class T>
void f(T a, T b)
{
    if(a > b) { ... }
}
```

在上述代码中，如果T的数据类型传入的是像Person这样的自定义数据类型，也无法正常运行

因此C++为了解决这种问题，提供模板的重载，可以为这些**特定的类型**提供**具体化的模板**

示例：

```
#include<iostream>
using namespace std;

#include <string>

class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
    string m_Name;
    int m_Age;
};

//普通函数模板
template<class T>
bool myCompare(T& a, T& b)
{
    if (a == b)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//具体化模板，显示具体化的原型和定意思以template<>开头，并通过名称来指出类型
//具体化模板优先于常规模板，如果是具体化的类型会比常规模板优先调用
template<> bool myCompare(Person &p1, Person &p2)
{
    if ( p1.m_Name == p2.m_Name && p1.m_Age == p2.m_Age )
    {
        return true;
    }
    else
    {
        return false;
    }
}

void test01()
{
    int a = 10;
    int b = 20;
    //内置数据类型可以直接使用通用的函数模板
    bool ret = myCompare(a, b);
    if (ret)
    {
        cout << "a == b " << endl;
    }
}
```

```

    }
    else
    {
        cout << "a != b " << endl;
    }
}

void test02()
{
    Person p1("Tom", 10);
    Person p2("Tom", 10);
    //自定义数据类型，不会调用普通的函数模板
    //可以创建具体化的Person数据类型的模板，用于特殊处理这个类型
    bool ret = myCompare(p1, p2);
    if (ret)
    {
        cout << "p1 == p2 " << endl;
    }
    else
    {
        cout << "p1 != p2 " << endl;
    }
}

int main() {
    test01();
    test02();
    system("pause");
    return 0;
}

```

总结：

- 利用具体化的模板，可以解决自定义类型的通用化
- 学习模板并不是为了写模板，而是在STL能够运用系统提供的模板

1.3 类模板

1.3.1 类模板语法

类模板作用：

- 建立一个通用类，类中的成员 数据类型可以不具体制定，用一个**虚拟的类型**来代表。

语法：

```
template<typename T>
类
```

解释：

template --- 声明创建模板

typename --- 表面其后面的符号是一种数据类型，可以用class代替

T --- 通用的数据类型，名称可以替换，通常为大写字母

示例：

```
#include <string>
//类模板
template<class NameType, class AgeType>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        this->mName = name;
        this->mAge = age;
    }
    void showPerson()
    {
        cout << "name: " << this->mName << " age: " << this->mAge << endl;
    }
public:
    NameType mName;
    AgeType mAge;
};

void test01()
{
    // 指定NameType 为string类型, AgeType 为 int类型
    Person<string, int>P1("孙悟空", 999);
    P1.showPerson();
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结：类模板和函数模板语法相似，在声明模板template后面加类，此类称为类模板

1.3.2 类模板与函数模板区别

类模板与函数模板区别主要有两点：

1. 类模板没有自动类型推导的使用方式（类模板只有显示类型）
2. 类模板在模板参数列表中可以有默认参数

示例：

```

#include <string>
//类模板
template<class NameType, class AgeType = int>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        this->mName = name;
        this->mAge = age;
    }
    void showPerson()
    {
        cout << "name: " << this->mName << " age: " << this->mAge << endl;
    }
public:
    NameType mName;
    AgeType mAge;
};

//1、类模板没有自动类型推导的使用方式
void test01()
{
    // Person p("孙悟空", 1000); // 错误 类模板使用时候，不可以用自动类型推导
    Person <string ,int>p("孙悟空", 1000); //必须使用显示指定类型的方式，使用类模板
    p.showPerson();
}

//2、类模板在模板参数列表中可以有默认参数
void test02()
{
    Person <string> p("猪八戒", 999); //类模板中的模板参数列表 可以指定默认参数
    p.showPerson();
}

int main() {
    test01();
    test02();
    system("pause");
    return 0;
}

```

总结：

- 类模板使用只能用显示指定类型方式
- 类模板中的模板参数列表可以有默认参数

1.3.3 类模板中成员函数创建时机

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：

- 普通类中的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

示例：

```

class Person1
{
public:
    void showPerson1()
    {
        cout << "Person1 show" << endl;
    }
};

class Person2
{
public:
    void showPerson2()
    {
        cout << "Person2 show" << endl;
    }
};

template<class T>
class MyClass
{
public:
    T obj;

    //类模板中的成员函数，并不是一开始就创建的，而是在模板调用时再生成

    void fun1() { obj.showPerson1(); }
    void fun2() { obj.showPerson2(); }

};

void test01()
{
    MyClass<Person1> m;

    m.fun1();

    //m.fun2(); //编译会出错，说明函数调用才会去创建成员函数
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

总结：类模板中的成员函数并不是一开始就创建的，在调用时才去创建

1.3.4 类模板对象做函数参数

学习目标：

- 类模板实例化出的对象，向函数传参的方式

一共有三种传入方式：

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

示例：

```
#include <string>
//类模板
template<class NameType, class AgeType = int>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        this->mName = name;
        this->mAge = age;
    }
    void showPerson()
    {
        cout << "name: " << this->mName << " age: " << this->mAge << endl;
    }
public:
    NameType mName;
    AgeType mAge;
};

//1、指定传入的类型
void printPerson1(Person<string, int> &p) //类模板中的对象做函数的参数传入的方法
{
    p.showPerson();
}
void test01()
{
    Person <string, int >p("孙悟空", 100);
    printPerson1(p);
}

//2、参数模板化
template <class T1, class T2>
void printPerson2(Person<T1, T2>&p)
{
    p.showPerson();
    cout << "T1的类型为: " << typeid(T1).name() << endl; //typeid可查看类型
    cout << "T2的类型为: " << typeid(T2).name() << endl;
}
void test02()
{
    Person <string, int >p("猪八戒", 90);
    printPerson2(p);
}

//3、整个类模板化
template<class T>
void printPerson3(T & p)
{
    cout << "T的类型为: " << typeid(T).name() << endl;
    p.showPerson();
}

void test03()
{
```

```
Person <string, int >p("唐僧", 30);
printPerson3(p);
}

int main() {
    test01();
    test02();
    test03();

    system("pause");
    return 0;
}
```

总结：

- 通过类模板创建的对象，可以有三种方式向函数中进行传参
- 使用比较广泛是第一种：指定传入的类型（后两种是函数模板混合类模板）

1.3.5 类模板与继承

当类模板碰到继承时，需要注意一下几点：

- 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，子类也需变为类模板

示例：

```

template<class T>
class Base
{
    T m;
};

//class Son:public Base //错误，c++编译需要给子类分配内存，必须知道父类中T的类型才可以向下继承
class Son :public Base<int> //必须指定一个类型
{
};

void test01()
{
    Son c;
}

//类模板继承类模板，可以用T2指定父类中的T类型
template<class T1, class T2>
class Son2 :public Base<T2>
{
public:
    Son2()
    {
        cout << typeid(T1).name() << endl;
        cout << typeid(T2).name() << endl;
        //类模板的成员函数只有在实例化的时候才会被调用
    }
};

void test02()
{
    Son2<int, char> child1;
}

int main() {
    test01();
    test02();
    system("pause");
    return 0;
}

```

总结：如果父类是类模板，子类需要指定出父类中T的数据类型

1.3.6 类模板成员函数类外实现

学习目标：能够掌握类模板中的成员函数类外实现

示例：

```

#include <string>

//类模板中成员函数类外实现
template<class T1, class T2>
class Person {
public:
    //成员函数类内声明
    Person(T1 name, T2 age);
    void showPerson();

public:
    T1 m_Name;
    T2 m_Age;
};

//构造函数 类外实现
template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age) {
    this->m_Name = name;
    this->m_Age = age;
}

//成员函数 类外实现
template<class T1, class T2>
void Person<T1, T2>::showPerson() {
    cout << "姓名: " << this->m_Name << " 年龄:" << this->m_Age << endl;
}

void test01()
{
    Person<string, int> p("Tom", 20);
    p.showPerson();
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：类模板中成员函数类外实现时，需要加上模板参数列表

1.3.7 类模板分文件编写

学习目标：

- 掌握类模板成员函数分文件编写产生的问题以及解决方式

问题：

- 类模板中成员函数创建时机是在调用阶段，导致分文件编写时链接不到

解决：

- 解决方式1：直接包含.cpp源文件
- 解决方式2：将声明和实现写到同一个文件中，并更改后缀名为.hpp，.hpp是约定的名称，并不是强制

示例：

person.hpp中代码：

```
#pragma once
#include <iostream>
using namespace std;
#include <string>

template<class T1, class T2>
class Person {
public:
    Person(T1 name, T2 age);
    void showPerson();
public:
    T1 m_Name;
    T2 m_Age;
};

//构造函数 类外实现
template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age) {
    this->m_Name = name;
    this->m_Age = age;
}

//成员函数 类外实现
template<class T1, class T2>
void Person<T1, T2>::showPerson() {
    cout << "姓名：" << this->m_Name << " 年龄：" << this->m_Age << endl;
}
```

类模板分文件编写.cpp中代码

```
#include<iostream>
using namespace std;

//#include "person.h" //注意类模板函数的创建时期是在调用阶段
#include "person.cpp" //解决方式1，包含cpp源文件

//解决方式2，将声明和实现写到一起，文件后缀名改为.hpp
#include "person.hpp"
void test01()
{
    Person<string, int> p("Tom", 10);
    p.showPerson();
}

int main() {
    test01();
    system("pause");
    return 0;
}
```

总结：主流的解决方式是第二种，将类模板成员函数写到一起，并将后缀名改为.hpp

1.3.8 类模板与友元

学习目标：

- 掌握类模板配合友元函数的类内和类外实现

全局函数类内实现 - 直接在类内声明友元即可

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

示例：

```
#include <string>

//2、全局函数配合友元  类外实现 - 先做函数模板声明，下方在做函数模板定义，在做友元
template<class T1, class T2> class Person;

//如果声明了函数模板，可以将实现写到后面，否则需要将实现体写到类的前面让编译器提前看到
//template<class T1, class T2> void printPerson2(Person<T1, T2> & p);

template<class T1, class T2>
void printPerson2(Person<T1, T2> & p)
{
    cout << "类外实现 ---- 姓名: " << p.m_Name << " 年龄: " << p.m_Age << endl;
}

template<class T1, class T2>
class Person
{
    //1、全局函数配合友元  类内实现
    friend void printPerson(Person<T1, T2> & p)
    {
        cout << "姓名: " << p.m_Name << " 年龄: " << p.m_Age << endl;
    }

    //全局函数配合友元  类外实现
    friend void printPerson2<>(Person<T1, T2> & p); //参数列表要写
}

public:

Person(T1 name, T2 age)
{
    this->m_Name = name;
    this->m_Age = age;
}

private:
T1 m_Name;
T2 m_Age;
};

//1、全局函数在类内实现
void test01()
{
    Person <string, int >p("Tom", 20);
    printPerson(p);
}

//2、全局函数在类外实现
void test02()
{
    Person <string, int >p("Jerry", 30);
    printPerson2(p);
```

```
}

int main() {
    //test01();

    test02();

    system("pause");

    return 0;
}
```

总结：建议全局函数做类内实现，用法简单，而且编译器可以直接识别

1.3.9 类模板案例

案例描述：实现一个通用的数组类，要求如下：

- 可以对内置数据类型以及自定义数据类型的数据进行存储
- 将数组中的数据存储到堆区
- 构造函数中可以传入数组的容量
- 提供对应的拷贝构造函数以及operator=防止浅拷贝问题
- 提供尾插法和尾删法对数组中的数据进行增加和删除
- 可以通过下标的方式访问数组中的元素
- 可以获取数组中当前元素个数和数组的容量

示例：

myArray.hpp中代码

```

#pragma once
#include <iostream>
using namespace std;

template<class T>
class MyArray
{
public:

    //构造函数
    MyArray(int capacity)
    {
        this->m_Capacity = capacity;
        this->m_Size = 0;
        pAddress = new T[this->m_Capacity];
    }

    //拷贝构造
    MyArray(const MyArray & arr)
    {
        this->m_Capacity = arr.m_Capacity;
        this->m_Size = arr.m_Size;
        this->pAddress = new T[this->m_Capacity];
        for (int i = 0; i < this->m_Size; i++)
        {
            //如果T为对象，而且还包含指针，必须需要重载 = 操作符，因为这个等号不是 构造 而是赋值
            // 普通类型可以直接= 但是指针类型需要深拷贝
            this->pAddress[i] = arr.pAddress[i];
        }
    }

    //重载= 操作符 防止浅拷贝问题
    MyArray& operator=(const MyArray& myarray) {

        if (this->pAddress != NULL) {
            delete[] this->pAddress;
            this->m_Capacity = 0;
            this->m_Size = 0;
        }

        this->m_Capacity = myarray.m_Capacity;
        this->m_Size = myarray.m_Size;
        this->pAddress = new T[this->m_Capacity];
        for (int i = 0; i < this->m_Size; i++) {
            this->pAddress[i] = myarray[i];
        }
        return *this;
    }

    //重载[] 操作符 arr[0]
    T& operator [](int index)
    {
        return this->pAddress[index]; //不考虑越界，用户自己去处理
    }
}

```

```

//尾插法
void Push_back(const T & val)
{
    if (this->m_Capacity == this->m_Size)
    {
        return;
    }
    this->pAddress[this->m_Size] = val;
    this->m_Size++;
}

//尾删法
void Pop_back()
{
    if (this->m_Size == 0)
    {
        return;
    }
    this->m_Size--;
}

//获取数组容量
int getCapacity()
{
    return this->m_Capacity;
}

//获取数组大小
int getSize()
{
    return this->m_Size;
}

//析构
~MyArray()
{
    if (this->pAddress != NULL)
    {
        delete[] this->pAddress;
        this->pAddress = NULL;
        this->m_Capacity = 0;
        this->m_Size = 0;
    }
}

private:
    T * pAddress; //指向一个堆空间，这个空间存储真正的数据
    int m_Capacity; //容量
    int m_Size; // 大小
};

```

类模板案例—数组类封装.cpp中

```

#include "myArray.hpp"
#include <string>

void printIntArray(MyArray<int>& arr) {
    for (int i = 0; i < arr.getSize(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

//测试内置数据类型
void test01()
{
    MyArray<int> array1(10);
    for (int i = 0; i < 10; i++)
    {
        array1.Push_back(i);
    }
    cout << "array1打印输出: " << endl;
    printIntArray(array1);
    cout << "array1的大小: " << array1.getSize() << endl;
    cout << "array1的容量: " << array1.getCapacity() << endl;

    cout << "-----" << endl;

    MyArray<int> array2(array1);
    array2.Pop_back();
    cout << "array2打印输出: " << endl;
    printIntArray(array2);
    cout << "array2的大小: " << array2.getSize() << endl;
    cout << "array2的容量: " << array2.getCapacity() << endl;
}

//测试自定义数据类型
class Person {
public:
    Person() {}
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
public:
    string m_Name;
    int m_Age;
};

void printPersonArray(MyArray<Person>& personArr)
{
    for (int i = 0; i < personArr.getSize(); i++) {
        cout << "姓名: " << personArr[i].m_Name << " 年龄: " << personArr[i].m_Age << endl
    }
}

void test02()

```

```

{
    //创建数组
    MyArray<Person> pArray(10);
    Person p1("孙悟空", 30);
    Person p2("韩信", 20);
    Person p3("妲己", 18);
    Person p4("王昭君", 15);
    Person p5("赵云", 24);

    //插入数据
    pArray.Push_back(p1);
    pArray.Push_back(p2);
    pArray.Push_back(p3);
    pArray.Push_back(p4);
    pArray.Push_back(p5);

    printPersonArray(pArray);

    cout << "pArray的大小: " << pArray.getSize() << endl;
    cout << "pArray的容量: " << pArray.getCapacity() << endl;
}

int main() {
    //test01();

    test02();

    system("pause");

    return 0;
}

```

总结：

能够利用所学知识点实现通用的数组

2 STL初识

2.1 STL的诞生

- 长久以来，软件界一直希望建立一种可重复利用的东西
- C++的**面向对象和泛型编程**思想，目的就是**复用性的提升**
- 大多情况下，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工作
- 为了建立数据结构和算法的一套标准，诞生了**STL**

2.2 STL基本概念

- STL(Standard Template Library, 标准模板库)
- STL 从广义上分为: 容器(container) 算法(algorithm) 迭代器(iterator)
- 容器和算法之间通过迭代器进行无缝连接。
- STL 几乎所有的代码都采用了模板类或者模板函数

2.3 STL六大组件

STL大体分为六大组件，分别是:容器、算法、迭代器、仿函数、适配器（配接器）、空间配置器

1. 容器：各种数据结构，如vector、list、deque、set、map等,用来存放数据。
2. 算法：各种常用的算法，如sort、find、copy、for_each等
3. 迭代器：扮演了容器与算法之间的胶合剂。
4. 仿函数：行为类似函数，可作为算法的某种策略。
5. 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。
6. 空间配置器：负责空间的配置与管理。

2.4 STL中容器、算法、迭代器

容器：置物之所也

STL容器就是将运用最广泛的一些数据结构实现出来

常用的数据结构：数组, 链表, 树, 栈, 队列, 集合, 映射表 等

这些容器分为序列式容器和关联式容器两种:

序列式容器:强调值的排序，序列式容器中的每个元素均有固定的位置。

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（有点类似于python的字典，没有顺序和索引）

算法：问题之解法也

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

算法分为**质变算法**和**非质变算法**。

质变算法：是指运算过程中会更改区间内的元素的内容。例如拷贝（12345->1122334455），替换，删除等等

非质变算法：是指运算过程中不会更改区间内的元素内容，例如查找、计数、遍历、寻找极值等等.

迭代器：容器和算法之间粘合剂

提供一种方法，使之能够依序寻访某个容器所含的各个元素，而又无需暴露该容器的内部表示方式。

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，初学阶段我们可以先理解迭代器为指针

迭代器种类：

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 <code>++</code> 、 <code>==</code> 、 <code>!=</code>
输出迭代器	对数据的只写访问	只写，支持 <code>++</code>
前向迭代器	读写操作，并能向前推进迭代器	读写，支持 <code>++</code> 、 <code>==</code> 、 <code>!=</code>
双向迭代器	读写操作，并能向前和向后操作	读写，支持 <code>++</code> 、 <code>--</code> ，
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 <code>++</code> 、 <code>--</code> 、 <code>[n]</code> 、 <code>-n</code> 、 <code><</code> 、 <code><=</code> 、 <code>></code> 、 <code>>=</code>

常用的容器中迭代器种类为双向迭代器，和随机访问迭代器

2.5 容器算法迭代器初识

了解STL中容器、算法、迭代器概念之后，我们利用代码感受STL的魅力

STL中最常用的容器为Vector，可以理解为数组，下面我们将学习如何向这个容器中插入数据、并遍历这个容器

2.5.1 vector存放内置数据类型

容器：`vector`

算法：`for_each`

迭代器：`vector<int>::iterator`

示例：

```
#include <vector>
#include <algorithm>

void MyPrint(int val)
{
    cout << val << endl;
}

void test01() {

    //创建vector容器对象，并且通过模板参数指定容器中存放的数据的类型
    vector<int> v;
    //向容器中放数据
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);

    //每一个容器都有自己的迭代器，迭代器是用来遍历容器中的元素
    //v.begin()返回迭代器，这个迭代器指向容器中第一个数据(返回的类似指针类型)
    //v.end()返回迭代器，这个迭代器指向容器元素的最后一个元素的下一个位置
    //vector<int>::iterator 拿到vector<int>这种容器的迭代器类型

    vector<int>::iterator pBegin = v.begin();
    vector<int>::iterator pEnd = v.end();

    //第一种遍历方式：
    while (pBegin != pEnd) {
        cout << *pBegin << endl;
        pBegin++;
    }

    //第二种遍历方式：
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        //it != v.end?? v.end不用写vector<int>::iterator ??吗
        cout << *it << endl;
    }
    cout << endl;

    //第三种遍历方式：
    //使用STL提供标准遍历算法 头文件 algorithm
    for_each(v.begin(), v.end(), MyPrint);
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

2.5.2 Vector存放自定义数据类型

学习目标：vector中存放自定义数据类型，并打印输出

示例：

```
#include <vector>
#include <string>
#include <algorithm>

//自定义数据类型
class Person {
public:
    Person(string name, int age) {
        mName = name;
        mAge = age;
    }
public:
    string mName;
    int mAge;
};

void myprint(Person* val)
{
    cout << "Name:" << val->mName << " Age:" << val->mAge << endl;
}

//存放对象
void test01() {

    vector<Person*> v;

    //创建数据
    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);
    Person p5("eee", 50);

    v.push_back(&p1);
    v.push_back(&p2);
    v.push_back(&p3);
    v.push_back(&p4);
    v.push_back(&p5);

    // for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
    //     cout << "Name:" << (*it).mName << " Age:" << (*it).mAge << endl;

    // }
    for_each(v.begin(), v.end(), myprint); //这里改成for_each算法实现，特别注意的是打印函数不要加
}

//放对象指针
void test02() {

    vector<Person*> v;

    //创建数据
    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
```

```

Person p4("ddd", 40);
Person p5("eee", 50);

v.push_back(&p1);
v.push_back(&p2);
v.push_back(&p3);
v.push_back(&p4);
v.push_back(&p5);

for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++) {
    Person * p = (*it);
    cout << "Name:" << p->mName << " Age:" << (*it)->mAge << endl;
}
}

int main() {

    test01();

    test02();

    system("pause");

    return 0;
}

```

2.5.3 Vector容器嵌套容器

学习目标：容器中嵌套容器，我们将所有数据进行遍历输出

示例：

```
#include <vector>
```

3 STL- 常用容器

3.1 string容器

3.1.1 string基本概念

本质：

- string是C++风格的字符串，而string本质上是一个类

string和char * 区别：

- `char *` 是一个指针
- `string`是一个类，类内部封装了`char*`，管理这个字符串，是一个`char*`型的容器。

特点：

`string` 类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

3.1.2 string构造函数

构造函数原型：

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

示例：

```
#include <string>
//string构造
void test01()
{
    string s1; //创建空字符串，调用无参构造函数
    cout << "str1 = " << s1 << endl;

    const char* str = "hello world";
    string s2(str); //把c_string转换成了string

    cout << "str2 = " << s2 << endl;

    string s3(s2); //调用拷贝构造函数
    cout << "str3 = " << s3 << endl;

    string s4(10, 'a');
    cout << "str3 = " << s3 << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结：`string`的多种构造方式没有可比性，灵活使用即可

3.1.3 string赋值操作

功能描述：

- 给string字符串进行赋值

赋值的函数原型：

- `string& operator=(const char* s); //char*类型字符串 赋值给当前的字符串`
- `string& operator=(const string &s); //把字符串s赋给当前的字符串`
- `string& operator=(char c); //字符赋值给当前的字符串`
- `string& assign(const char *s); //把字符串s赋给当前的字符串`
- `string& assign(const char *s, int n); //把字符串s的前n个字符赋给当前的字符串`
- `string& assign(const string &s); //把字符串s赋给当前字符串`
- `string& assign(int n, char c); //用n个字符c赋给当前字符串`

示例：

```

//赋值
void test01()
{
    string str1;
    str1 = "hello world";
    cout << "str1 = " << str1 << endl;

    string str2;
    str2 = str1;
    cout << "str2 = " << str2 << endl;

    string str3;
    str3 = 'a';
    cout << "str3 = " << str3 << endl;

    string str4;
    str4.assign("hello c++");
    cout << "str4 = " << str4 << endl;

    string str5;
    str5.assign("hello c++",5);
    cout << "str5 = " << str5 << endl;

    string str6;
    str6.assign(str5);
    cout << "str6 = " << str6 << endl;

    string str7;
    str7.assign(5, 'x');
    cout << "str7 = " << str7 << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

总结：

string的赋值方式很多， operator= 这种方式是比较实用的

3.1.4 string字符串拼接

功能描述：

- 实现在字符串末尾拼接字符串

函数原型：

- `string& operator+=(const char* str); //重载+=操作符`
- `string& operator+=(const char c); //重载+=操作符`
- `string& operator+=(const string& str); //重载+=操作符`
- `string& append(const char *s); //把字符串s连接到当前字符串结尾`
- `string& append(const char *s, int n); //把字符串s的前n个字符连接到当前字符串结尾`
- `string& append(const string &s); //同operator+=(const string& str)`
- `string& append(const string &s, int pos, int n); //字符串s中从pos开始的n个字符连接到字符串结尾`

示例：

```
//字符串拼接
void test01()
{
    string str1 = "我";

    str1 += "爱玩游戏";

    cout << "str1 = " << str1 << endl;

    str1 += ':';

    cout << "str1 = " << str1 << endl;

    string str2 = "LOL DNF";

    str1 += str2;

    cout << "str1 = " << str1 << endl;

    string str3 = "I";
    str3.append(" love ");
    str3.append("game abcde", 4);
    //str3.append(str2);
    str3.append(str2, 4, 3); // 从下标4位置开始，截取3个字符，拼接到字符串末尾
    cout << "str3 = " << str3 << endl;
}
int main() {

    test01();

    system("pause");

    return 0;
}
```

总结：字符串拼接的重载版本很多，初学阶段记住几种即可

3.1.5 string查找和替换

功能描述：

- 查找：查找指定字符串是否存在
- 替换：在指定的位置替换字符串

函数原型：

- int find(const string& str, int pos = 0) const; //查找str第一次出现位置,从pos开始查找, pos默认为0
- int find(const char* s, int pos = 0) const; //查找s第一次出现位置,从pos开始查找
- int find(const char* s, int pos, int n) const; //从pos位置查找s的前n个字符第一次位置
- int find(const char c, int pos = 0) const; //查找字符c第一次出现位置
- int rfind(const string& str, int pos = npos) const; //查找str最后一次位置,从pos开始查找
- int rfind(const char* s, int pos = npos) const; //查找s最后一次出现位置,从pos开始查找
- int rfind(const char* s, int pos, int n) const; //从pos查找s的前n个字符最后一次位置
- int rfind(const char c, int pos = 0) const; //查找字符c最后一次出现位置
- string& replace(int pos, int n, const string& str); //替换从pos开始n个字符为字符串str
- string& replace(int pos, int n, const char* s); //替换从pos开始的n个字符为字符串s

示例：

```

//查找和替换
void test01()
{
    //查找
    string str1 = "abcdefgde";

    int pos = str1.find("de");           //找不到的情况下，pos返回-1

    if (pos == -1)
    {
        cout << "未找到" << endl;
    }
    else
    {
        cout << "pos = " << pos << endl;
    }

    pos = str1.rfind("de");

    cout << "pos = " << pos << endl;
}

void test02()
{
    //替换
    string str1 = "abcdefgde";
    str1.replace(1, 3, "1111");

    cout << "str1 = " << str1 << endl;
}

int main() {

    //test01();
    //test02();

    system("pause");

    return 0;
}

```

总结：

- find查找是从左往右， rfind从右往左
- find找到字符串后返回查找的第一个字符位置，找不到返回-1
- replace在替换时，要指定从哪个位置起，多少个字符，替换成什么样的字符串

3.1.6 string字符串比较

功能描述：

- 字符串之间的比较

比较方式：

- 字符串比较是按字符的ASCII码进行对比

= 返回 0

> 返回 1

< 返回 -1

函数原型：

- int compare(const string &s) const; //与字符串s比较
- int compare(const char *s) const; //与字符串s比较

示例：

```
//字符串比较
void test01()
{
    string s1 = "hello";
    string s2 = "aello";

    int ret = s1.compare(s2);

    if (ret == 0) {
        cout << "s1 等于 s2" << endl;
    }
    else if (ret > 0)
    {
        cout << "s1 大于 s2" << endl;
    }
    else
    {
        cout << "s1 小于 s2" << endl;
    }
}

int main() {
    test01();
    system("pause");
    return 0;
}
```

总结：字符串对比主要是用于比较两个字符串是否相等，判断谁大谁小的意义并不是很大

3.1.7 string字符存取

string中单个字符存取方式有两种

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

示例：

```
void test01()
{
    string str = "hello world";

    for (int i = 0; i < str.size(); i++) //size函数访问字符串长度
    {
        cout << str[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < str.size(); i++)
    {
        cout << str.at(i) << " ";
    }
    cout << endl;

    //字符修改
    str[0] = 'x';
    str.at(1) = 'x';
    cout << str << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：string字符串中单个字符存取有两种方式，利用 [] 或 at

3.1.8 string插入和删除

功能描述：

- 对string字符串进行插入和删除字符操作

函数原型：

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

示例：

```
//字符串插入和删除
void test01()
{
    string str = "hello";
    str.insert(1, "111");
    cout << str << endl;

    str.erase(1, 3); //从1号位置开始3个字符
    cout << str << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：插入和删除的起始下标都是从0开始

3.1.9 string子串

功能描述：

- 从字符串中获取想要的子串

函数原型：

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

示例：

```

//子串
void test01()
{
    string str = "abcdefg";
    string subStr = str.substr(1, 3);
    cout << "subStr = " << subStr << endl;

    string email = "hello@sina.com";
    int pos = email.find("@");
    string username = email.substr(0, pos);
    cout << "username: " << username << endl;
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：灵活的运用求子串功能，可以在实际开发中获取有效的信息

3.2 vector容器

3.2.1 vector基本概念

功能：

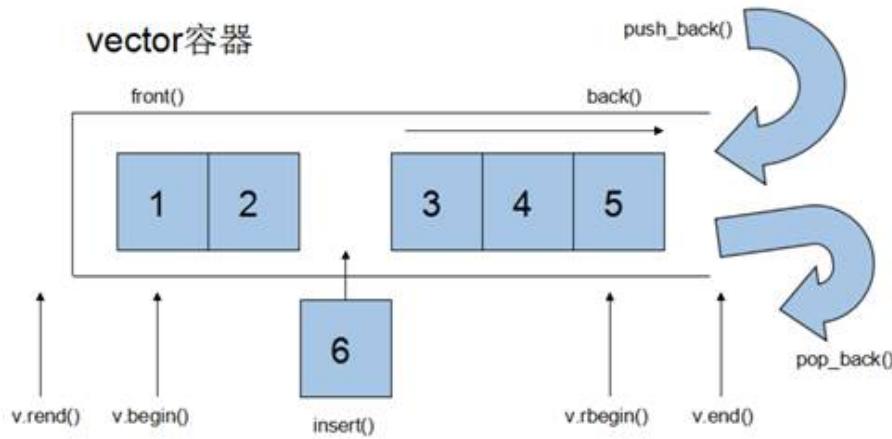
- vector数据结构和**数组非常相似**，也称为**单端数组**

vector与普通数组区别：

- 不同之处在于数组是静态空间，而vector可以**动态扩展**

动态扩展：

- 并不是在原空间之后续接新空间，而是找更大的内存空间(大小并不固定)，然后将原数据拷贝新空间，释放原空间



- vector容器的迭代器是支持随机访问的迭代器（最强悍的迭代器 可以跳着访问）

3.2.2 vector构造函数

功能描述：

- 创建vector容器

函数原型：

- `vector<T> v;` //采用模板实现类实现， 默认构造函数
- `vector(v.begin(), v.end());` //将v[begin(), end())区间中的元素拷贝给本身。前闭后开区间
- `vector(n, elem);` //构造函数将n个elem拷贝给本身。
- `vector(const vector &vec);` //拷贝构造函数。

示例：

```

#include <vector>

void printVector(vector<int>& v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    vector<int> v1; //无参构造
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);

    vector<int> v2(v1.begin(), v1.end());
    printVector(v2);

    vector<int> v3(10, 100);
    printVector(v3);

    vector<int> v4(v3);
    printVector(v4);
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：vector的多种构造方式没有可比性，灵活使用即可，拷贝构造用的较多。

3.2.3 vector赋值操作

功能描述：

- 给vector容器进行赋值

函数原型：

- vector& operator=(const vector &vec); //重载等号操作符
- assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
- assign(n, elem); //将n个elem拷贝赋值给本身。

示例：

```
#include <vector>

void printVector(vector<int>& v) {

    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//赋值操作
void test01()
{
    vector<int> v1; //无参构造
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);

    vector<int> v2;
    v2 = v1;
    printVector(v2);

    vector<int> v3;
    v3.assign(v1.begin(), v1.end());
    printVector(v3);

    vector<int> v4;
    v4.assign(10, 100);
    printVector(v4);
}

int main() {
    test01();
    system("pause");
    return 0;
}
```

总结：vector赋值方式比较简单，使用operator=，或者assign都可以

3.2.4 vector容量和大小

功能描述：

- 对vector容器的容量和大小操作

函数原型：

- `empty();` //判断容器是否为空
- `capacity();` //容器的容量
- `size();` //返回容器中元素的个数
- `resize(int num);` //重新指定容器的长度为num，若容器变长，则以默认值填充新位置。默认值为0来填充
//如果容器变短，则末尾超出容器长度的元素被删除。
- `resize(int num, elem);` //重新指定容器的长度为num，若容器变长，则以elem值填充新位置。
//如果容器变短，则末尾超出容器长度的元素被删除

示例：

```

#include <vector>

void printVector(vector<int>& v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    vector<int> v1;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);
    if (v1.empty())
    {
        cout << "v1为空" << endl;
    }
    else
    {
        cout << "v1不为空" << endl;
        cout << "v1的容量 = " << v1.capacity() << endl;
        cout << "v1的大小 = " << v1.size() << endl;
    }

    //resize 重新指定大小 , 若指定的更大, 默认用0填充新位置, 可以利用重载版本替换默认填充
    v1.resize(15, 10);
    printVector(v1);

    //resize 重新指定大小 , 若指定的更小, 超出部分元素被删除
    v1.resize(5);
    printVector(v1);
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 判断是否为空 --- empty
- 返回元素个数 --- size
- 返回容器容量 --- capacity

- 重新指定大小 --- `resize`

3.2.5 vector插入和删除

功能描述：

- 对vector容器进行插入、删除操作

函数原型：

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

示例：

```

#include <vector>

void printVector(vector<int>& v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//插入和删除
void test01() {
    vector<int> v1;
    //尾插
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);
    printVector(v1);
    //尾删
    v1.pop_back();
    printVector(v1);
    //插入
    v1.insert(v1.begin(), 100);
    printVector(v1);

    v1.insert(v1.begin(), 2, 1000);
    printVector(v1);

    //删除
    v1.erase(v1.begin());
    printVector(v1);

    //清空
    v1.erase(v1.begin(), v1.end());
    v1.clear(); //和上一行代码效果相同
    printVector(v1);
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 尾插 --- push_back

- 尾删 --- `pop_back`
- 插入 --- `insert` (位置迭代器)
- 删除 --- `erase` (位置迭代器)
- 清空 --- `clear`

3.2.6 vector数据存取

功能描述：

- 对vector中的数据的存取操作

函数原型：

- `at(int idx);` //返回索引idx所指的数据
- `operator[];` //返回索引idx所指的数据
- `front();` //返回容器中第一个数据元素
- `back();` //返回容器中最后一个数据元素

示例：

```

#include <vector>

void test01()
{
    vector<int>v1;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }

    for (int i = 0; i < v1.size(); i++)
    {
        cout << v1[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < v1.size(); i++)
    {
        cout << v1.at(i) << " ";
    }
    cout << endl;

    cout << "v1的第一个元素为: " << v1.front() << endl;
    cout << "v1的最后一个元素为: " << v1.back() << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 除了用迭代器获取vector容器中元素，[]和at也可以
- front返回容器第一个元素
- back返回容器最后一个元素

3.2.7 vector互换容器

功能描述：

- 实现两个容器内元素进行互换

函数原型：

- swap(vec); // 将vec与本身的元素互换

示例：

```
#include <vector>

void printVector(vector<int>& v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    vector<int>v1;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);

    vector<int>v2;
    for (int i = 10; i > 0; i--)
    {
        v2.push_back(i);
    }
    printVector(v2);

    //互换容器
    cout << "互换后" << endl;
    v1.swap(v2);
    printVector(v1);
    printVector(v2);
}

void test02()
{
    vector<int> v;
    for (int i = 0; i < 100000; i++) {
        v.push_back(i);
    }

    cout << "v的容量为：" << v.capacity() << endl;
    cout << "v的大小为：" << v.size() << endl;

    v.resize(3);

    cout << "v的容量为：" << v.capacity() << endl;
    cout << "v的大小为：" << v.size() << endl;

    //收缩内存
    vector<int>(v).swap(v); //匿名对象 在调用后就会立即回收内存 vector<int>(v)是匿名对象

    cout << "v的容量为：" << v.capacity() << endl;
    cout << "v的大小为：" << v.size() << endl;
}
```

```
int main() {  
    test01();  
    test02();  
    system("pause");  
    return 0;  
}
```

总结：swap可以使两个容器互换，可以达到实用的收缩内存效果

3.2.8 vector预留空间

功能描述：

- 减少vector在动态扩展容量时的扩展次数

函数原型：

- `reserve(int len);` //容器预留len个元素长度，预留位置不初始化，元素不可访问。

示例：

```

#include <vector>

void test01()
{
    vector<int> v;

    //预留空间
    v.reserve(100000); //如果没有这一行代码预留指定的空间 下面的num可能是很多次，重新开辟很多次空间。

    int num = 0;
    int* p = NULL;
    for (int i = 0; i < 100000; i++) {
        v.push_back(i);
        if (p != &v[0]) {
            p = &v[0];
            num++;
        }
    }

    cout << "num:" << num << endl;
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：如果数据量较大，可以一开始利用reserve预留空间

3.3 deque容器

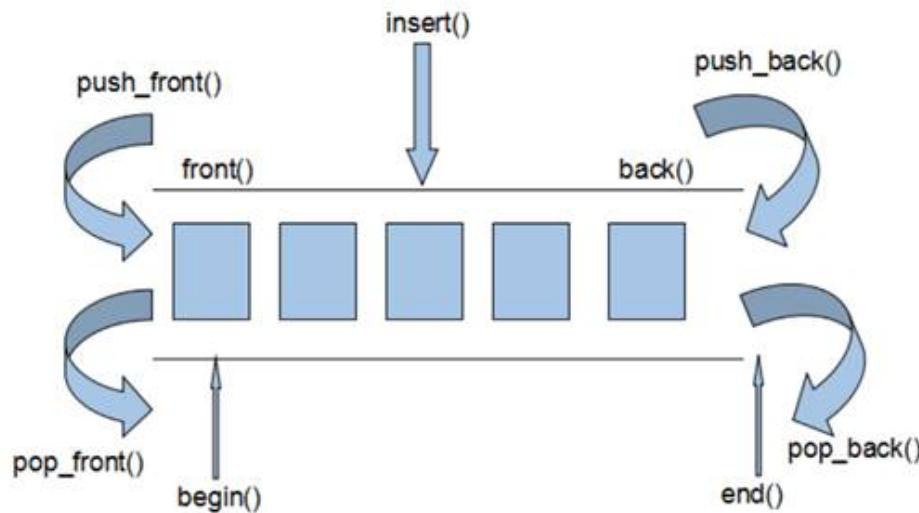
3.3.1 deque容器基本概念

功能：

- 双端数组，可以对头端进行插入删除操作

deque与vector区别：

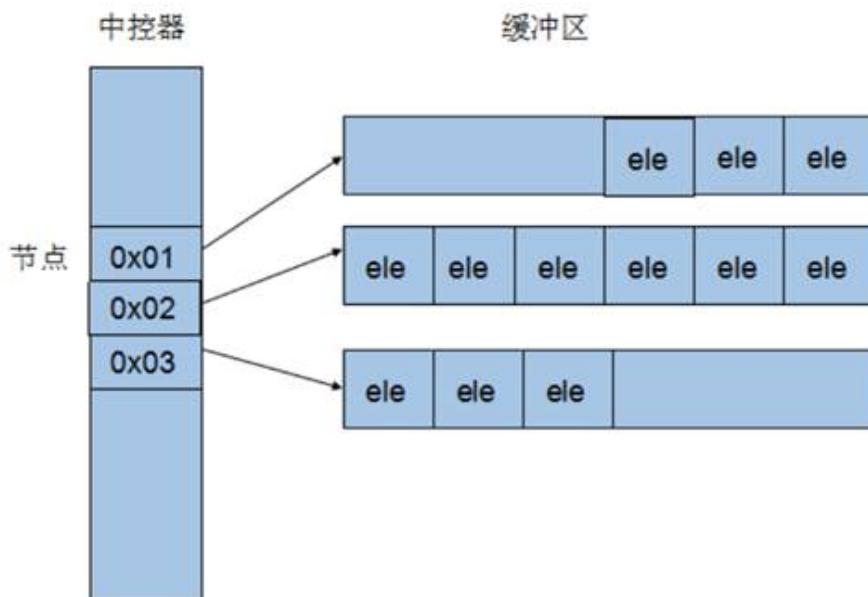
- vector对于头部的插入删除效率低，数据量越大，效率越低（比如10万个数据 要先把10万个数据向后复制一个空间再把第一个空间赋值）
- deque相对而言，对头部的插入删除速度比vector快
- vector访问元素时的速度会比deque快,这和两者内部实现有关



deque内部工作原理:

deque内部有个**中控器**, 维护每段缓冲区中的内容, 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, 使得使用deque时像一片连续的内存空间 (如果前后空间不够会再找一小段作为储存空间, 然后把这一小段地址记录再中控器里, 它访问的速度不如vector快)



- deque容器的迭代器也是支持随机访问的

3.3.2 deque构造函数

功能描述:

- deque容器构造

函数原型:

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

示例：

```
#include <deque>

void printDeque(const deque<int>& d) //限制容器里的数据为可读状态
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) { //const_iterator
        cout << *it << " ";
    }
    cout << endl;
}

//deque构造
void test01() {

    deque<int> d1; //无参构造函数
    for (int i = 0; i < 10; i++)
    {
        d1.push_back(i);
    }
    printDeque(d1);
    deque<int> d2(d1.begin(), d1.end());
    printDeque(d2);

    deque<int> d3(10, 100);
    printDeque(d3);

    deque<int> d4 = d3;
    printDeque(d4);
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：deque容器和vector容器的构造方式几乎一致，灵活使用即可

3.3.3 deque赋值操作

功能描述：

- 给deque容器进行赋值

函数原型：

- `deque& operator=(const deque &deq);` //重载等号操作符
- `assign(beg, end);` //将[beg, end)区间中的数据拷贝赋值给本身。
- `assign(n, elem);` //将n个elem拷贝赋值给本身。

示例：

```
#include <deque>

void printDeque(const deque<int>& d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//赋值操作
void test01()
{
    deque<int> d1;
    for (int i = 0; i < 10; i++)
    {
        d1.push_back(i);
    }
    printDeque(d1);

    deque<int> d2;
    d2 = d1;
    printDeque(d2);

    deque<int> d3;
    d3.assign(d1.begin(), d1.end());
    printDeque(d3);

    deque<int> d4;
    d4.assign(10, 100);
    printDeque(d4);
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：deque赋值操作也与vector相同，需熟练掌握

3.3.4 deque大小操作

功能描述：

- 对deque容器的大小进行操作

函数原型：

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

deque容器没有容量的概念，可以无限扩展，这一点跟vector容器不同

示例：

```

#include <deque>

void printDeque(const deque<int>& d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//大小操作
void test01()
{
    deque<int> d1;
    for (int i = 0; i < 10; i++)
    {
        d1.push_back(i);
    }
    printDeque(d1);

    //判断容器是否为空
    if (d1.empty()) {
        cout << "d1为空!" << endl;
    }
    else {
        cout << "d1不为空!" << endl;
        //统计大小
        cout << "d1的大小为：" << d1.size() << endl;
    }

    //重新指定大小
    d1.resize(15, 1);
    printDeque(d1);

    d1.resize(5);
    printDeque(d1);
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- deque没有容量的概念
- 判断是否为空 --- empty
- 返回元素个数 --- size

- 重新指定个数 --- `resize`

3.3.5 deque 插入和删除

功能描述：

- 向deque容器中插入和删除数据

函数原型：

两端插入操作：

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

指定位置操作：

- `insert(pos, elem);` //在pos位置插入一个elem元素的拷贝，返回新数据的位置。
- `insert(pos, n, elem);` //在pos位置插入n个elem数据，无返回值。
- `insert(pos, beg, end);` //在pos位置插入[beg,end)区间的数据，无返回值。
- `clear();` //清空容器的所有数据
- `erase(beg, end);` //删除[beg,end)区间的数据，返回下一个数据的位置。
- `erase(pos);` //删除pos位置的数据，返回下一个数据的位置。

示例：

```
#include <deque>

void printDeque(const deque<int>& d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//两端操作
void test01()
{
    deque<int> d;
    //尾插
    d.push_back(10);
    d.push_back(20);
    //头插
    d.push_front(100);
    d.push_front(200);

    printDeque(d);

    //尾删
    d.pop_back();
    //头删
    d.pop_front();
    printDeque(d);
}

//插入
void test02()
{
    deque<int> d;
    d.push_back(10);
    d.push_back(20);
    d.push_front(100);
    d.push_front(200);
    printDeque(d);

    d.insert(d.begin(), 1000);
    printDeque(d);

    d.insert(d.begin(), 2, 10000);
    printDeque(d);

    deque<int> d2;
    d2.push_back(1);
    d2.push_back(2);
    d2.push_back(3);

    d.insert(d.begin(), d2.begin(), d2.end());
    printDeque(d);
}
```

```

//删除
void test03()
{
    deque<int> d;
    d.push_back(10);
    d.push_back(20);
    d.push_front(100);
    d.push_front(200);
    printDeque(d);

    d.erase(d.begin());
    printDeque(d);

    d.erase(d.begin(), d.end());
    d.clear();
    printDeque(d);
}

int main() {
    //test01();

    //test02();

    test03();

    system("pause");

    return 0;
}

```

总结：

- 插入和删除提供的位置是迭代器！（不要提供索引值，索引值是不可以的）
- 尾插 --- push_back
- 尾删 --- pop_back
- 头插 --- push_front
- 头删 --- pop_front

3.3.6 deque 数据存取

功能描述：

- 对deque 中的数据的存取操作

函数原型：

- at(int idx); //返回索引idx所指的数据
- operator[]; //返回索引idx所指的数据

- `front();` //返回容器中第一个数据元素
- `back();` //返回容器中最后一个数据元素

示例：

```
#include <deque>

void printDeque(const deque<int>& d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//数据存取
void test01()
{
    deque<int> d;
    d.push_back(10);
    d.push_back(20);
    d.push_front(100);
    d.push_front(200);

    for (int i = 0; i < d.size(); i++) {
        cout << d[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < d.size(); i++) {
        cout << d.at(i) << " ";
    }
    cout << endl;

    cout << "front:" << d.front() << endl;

    cout << "back:" << d.back() << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：

- 除了用迭代器获取deque容器中元素，[]和at也可以
- front返回容器第一个元素
- back返回容器最后一个元素

3.3.7 deque 排序

功能描述：

- 利用算法实现对deque容器进行排序

算法：

- sort(iterator beg, iterator end) //对beg和end区间内元素进行排序,注意这里是迭代器

示例：

```
#include <deque>
#include <algorithm>

void printDeque(const deque<int>& d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    deque<int> d;
    d.push_back(10);
    d.push_back(20);
    d.push_front(100);
    d.push_front(200);

    printDeque(d);
    sort(d.begin(), d.end());
    printDeque(d);
}

int main() {
    test01();
    system("pause");
    return 0;
}
```

总结：sort算法非常实用，使用时包含头文件 algorithm即可

3.4 案例-评委打分

3.4.1 案例描述

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

3.4.2 实现步骤

1. 创建五名选手，放到vector中
2. 遍历vector容器，取出来每一个选手，执行for循环，可以把10个评分打分存到deque容器中
3. sort算法对deque容器中分数排序，去除最高和最低分
4. deque容器遍历一遍，累加总分
5. 获取平均分

示例代码：

```
//选手类
class Person
{
public:
    Person(string name, int score)
    {
        this->m_Name = name;
        this->m_Score = score;
    }

    string m_Name; //姓名
    int m_Score; //平均分
};

void createPerson(vector<Person>&v)
{
    string nameSeed = "ABCDE";
    for (int i = 0; i < 5; i++)
    {
        string name = "选手";
        name += nameSeed[i];

        int score = 0;

        Person p(name, score);

        //将创建的person对象 放入到容器中
        v.push_back(p);
    }
}

//打分
void setScore(vector<Person>&v)
{
    for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
    {
        //将评委的分数 放入到deque容器中
        deque<int>d;
        for (int i = 0; i < 10; i++)
        {
            int score = rand() % 41 + 60; // 60 ~ 100
            d.push_back(score);
        }

        //cout << "选手: " << it->m_Name << " 打分: " << endl;
        //for (deque<int>::iterator dit = d.begin(); dit != d.end(); dit++)
        //{
        //    cout << *dit << " ";
        //}
        //cout << endl;

        //排序
        sort(d.begin(), d.end());

        //去除最高和最低分
    }
}
```

```

        d.pop_back();
        d.pop_front();           //需要对头端进行操作时用deque容器比较合适

        //取平均分
        int sum = 0;
        for (deque<int>::iterator dit = d.begin(); dit != d.end(); dit++)
        {
            sum += *dit; //累加每个评委的分数
        }

        int avg = sum / d.size();

        //将平均分 赋值给选手身上
        it->m_Score = avg;
    }

}

void showScore(vector<Person>&v)
{
    for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << "姓名: " << it->m_Name << " 平均分: " << it->m_Score << endl;
    }
}

int main() {

    //随机数种子
    srand((unsigned int)time(NULL));

    //1、创建5名选手
    vector<Person>v; //存放选手容器
    createPerson(v);

    //测试
    //for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
    //{
    //    cout << "姓名: " << (*it).m_Name << " 分数: " << (*it).m_Score << endl;
    //}

    //2、给5名选手打分
    setScore(v);

    //3、显示最后得分
    showScore(v);

    system("pause");
}

return 0;
}

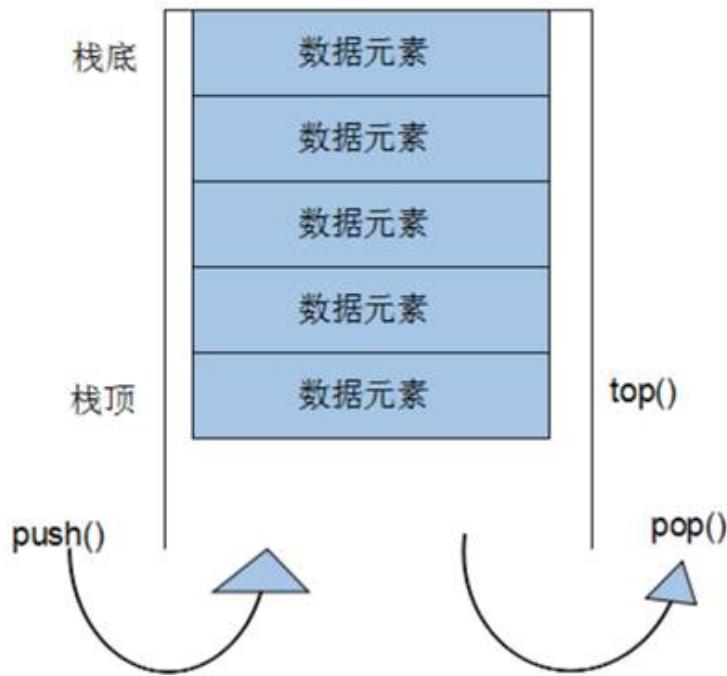
```

总结：选取不同的容器操作数据，可以提升代码的效率

3.5 stack容器

3.5.1 stack 基本概念

概念：stack是一种先进后出(First In Last Out,FILO)的数据结构，它只有一个出口（需要下面的元素都出去了才能出去）



栈中只有顶端的元素才可以被外界使用，因此栈不允许有遍历行为

栈中进入数据称为 --- 入栈 push

栈中弹出数据称为 --- 出栈 pop

生活中的栈：





3.5.2 stack 常用接口

功能描述：栈容器常用的对外接口

构造函数：

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

赋值操作：

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

数据存取：

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

大小操作：

- `empty();` //判断堆栈是否为空
- `size();` //返回栈的大小（入栈的时候进行记录的）

示例：

```

#include <stack>

//栈容器常用接口
void test01()
{
    //创建栈容器 栈容器必须符合先进后出
    stack<int> s;

    //向栈中添加元素，叫做 压栈 入栈
    s.push(10);
    s.push(20);
    s.push(30);

    while (!s.empty()) {
        //输出栈顶元素
        cout << "栈顶元素为: " << s.top() << endl;
        //弹出栈顶元素
        s.pop();
    }
    cout << "栈的大小为: " << s.size() << endl;
}

int main() {

    test01();

    system("pause");
    return 0;
}

```

总结：

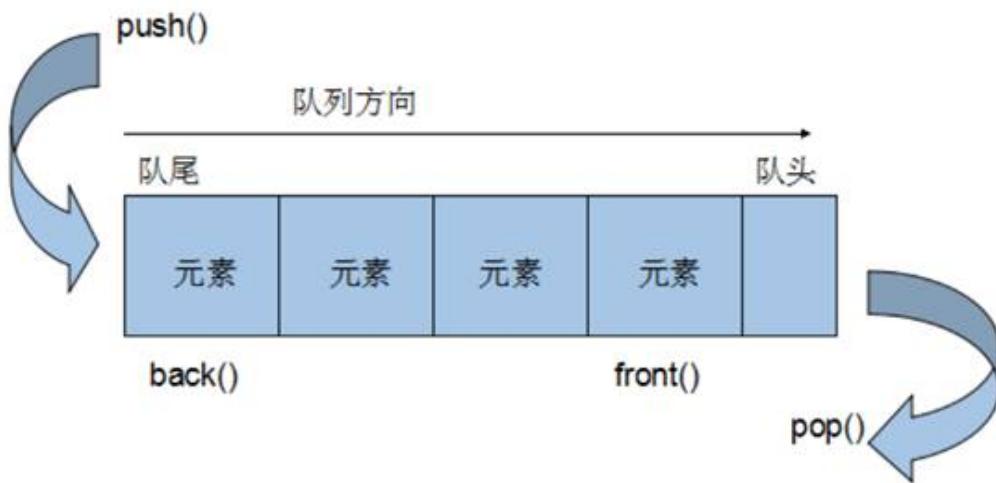
- 入栈 --- push
- 出栈 --- pop
- 返回栈顶 --- top
- 判断栈是否为空 --- empty
- 返回栈大小 --- size

3.6 queue 容器

3.6.1 queue 基本概念

概念：Queue是一种先进先出(First In First Out,FIFO)的数据结构，它有两个出口

队头只能弹出数据，队尾只能进入数据



队列容器允许从一端新增元素，从另一端移除元素

队列中只有队头和队尾才可以被外界使用，因此队列不允许有遍历行为

队列中进数据称为 --- 入队 push

队列中出数据称为 --- 出队 pop

生活中的队列：



3.6.2 queue 常用接口

功能描述：栈容器常用的对外接口

构造函数：

- `queue<T> que;` //queue采用模板类实现，queue对象的默认构造形式

- `queue(const queue &que); //拷贝构造函数`

赋值操作：

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

数据存取：

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

大小操作：

- `empty(); //判断堆栈是否为空`
- `size(); //返回栈的大小`

示例：

```
#include <queue>
#include <string>
class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }

    string m_Name;
    int m_Age;
};

void test01() {

    //创建队列
    queue<Person> q;

    //准备数据
    Person p1("唐僧", 30);
    Person p2("孙悟空", 1000);
    Person p3("猪八戒", 900);
    Person p4("沙僧", 800);

    //向队列中添加元素 入队操作
    q.push(p1);
    q.push(p2);
    q.push(p3);
    q.push(p4);

    //队列不提供迭代器，更不支持随机访问
    while (!q.empty()) {
        //输出队头元素
        cout << "队头元素-- 姓名: " << q.front().m_Name
        << " 年龄: " << q.front().m_Age << endl;

        cout << "队尾元素-- 姓名: " << q.back().m_Name
        << " 年龄: " << q.back().m_Age << endl;

        cout << endl;
        //弹出队头元素
        q.pop();
    }

    cout << "队列大小为: " << q.size() << endl;
}

int main() {
    test01();

    system("pause");
}
```

```
    return 0;  
}
```

总结：

- 入队 --- push
- 出队 --- pop
- 返回队头元素 --- front
- 返回队尾元素 --- back
- 判断队是否为空 --- empty
- 返回队列大小 --- size

3.7 list容器

3.7.1 list基本概念

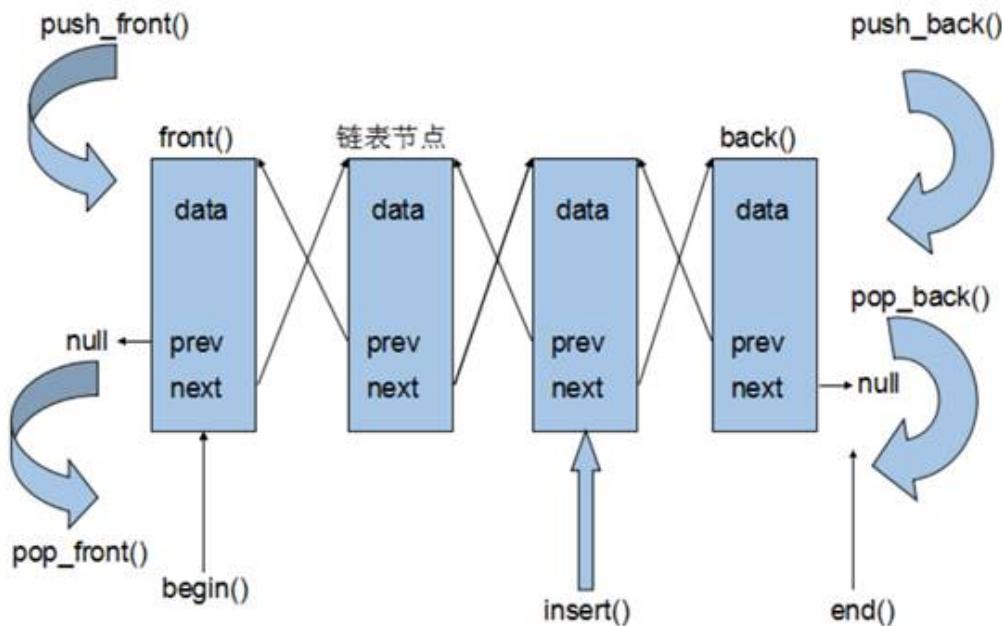
功能：将数据进行链式存储

链表 (list) 是一种物理存储单元上非连续的存储结构，数据元素的逻辑顺序是通过链表中的指针链接实现的

链表的组成：链表由一系列**结点**组成

结点的组成：一个是存储数据元素的**数据域**，另一个是存储下一个结点地址的**指针域**

STL中的链表是一个双向循环链表（每个节点即记录下一个节点位置又记录上一个节点的位置）



由于链表的存储方式并不是连续的内存空间，因此链表list中的迭代器只支持前移和后移，属于**双向迭代器**

list的优点：

- 采用动态存储分配，不会造成内存浪费和溢出

- 链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素

list的缺点：

- 链表灵活，但是空间(指针域) 和 时间（遍历）额外耗费较大

List有一个重要的性质，插入操作和删除操作都不会造成原有list迭代器的失效，这在vector是不成立的。

总结：STL中**List**和**vector**是两个最常被使用的容器，各有优缺点

3.7.2 list构造函数

功能描述：

- 创建list容器

函数原型：

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

示例：

```

#include <list>

void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    printList(L1);

    list<int>L2(L1.begin(),L1.end());
    printList(L2);

    list<int>L3(L2);
    printList(L3);

    list<int>L4(10, 1000);
    printList(L4);
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：list构造方式同其他几个STL常用容器，熟练掌握即可

3.7.3 list 赋值和交换

功能描述：

- 给list容器进行赋值，以及交换list容器

函数原型：

- assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
- assign(n, elem); //将n个elem拷贝赋值给本身。
- list& operator=(const list &lst); //重载等号操作符

- swap(lst); //将lst与本身的元素互换。

示例：

```
#include <list>

void printList(const list<int>& L) {

    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//赋值和交换
void test01()
{
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);
    printList(L1);

    //赋值
    list<int>L2;
    L2 = L1;
    printList(L2);

    list<int>L3;
    L3.assign(L2.begin(), L2.end());
    printList(L3);

    list<int>L4;
    L4.assign(10, 100);
    printList(L4);

}

//交换
void test02()
{

    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    list<int>L2;
    L2.assign(10, 100);

    cout << "交换前: " << endl;
    printList(L1);
    printList(L2);

    cout << endl;

    L1.swap(L2);

}
```

```
cout << "交换后: " << endl;
printList(L1);
printList(L2);

}

int main() {
    //test01();
    test02();
    system("pause");
    return 0;
}
```

总结：list赋值和交换操作能够灵活运用即可

3.7.4 list 大小操作

功能描述：

- 对list容器的大小进行操作

函数原型：

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

示例：

```

#include <list>

void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//大小操作
void test01()
{
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    if (L1.empty())
    {
        cout << "L1为空" << endl;
    }
    else
    {
        cout << "L1不为空" << endl;
        cout << "L1的大小为: " << L1.size() << endl;
    }

    //重新指定大小
    L1.resize(10);
    printList(L1);

    L1.resize(2);
    printList(L1);
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 判断是否为空 --- empty
- 返回元素个数 --- size
- 重新指定个数 --- resize

3.7.5 list 插入和删除

功能描述：

- 对list容器进行数据的插入和删除

函数原型：

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

示例：

```
#include <list>

void printList(const list<int>& L) {

    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

//插入和删除
void test01()
{
    list<int> L;
    //尾插
    L.push_back(10);
    L.push_back(20);
    L.push_back(30);
    //头插
    L.push_front(100);
    L.push_front(200);
    L.push_front(300);

    printList(L);

    //尾删
    L.pop_back();
    printList(L);

    //头删
    L.pop_front();
    printList(L);

    //插入
    list<int>::iterator it = L.begin();
    L.insert(++it, 1000);
    printList(L);

    //删除
    it = L.begin();
    L.erase(++it);
    printList(L);

    //移除
    L.push_back(10000);
    L.push_back(10000);
    L.push_back(10000);
    printList(L);
    L.remove(10000);
    printList(L);

    //清空
    L.clear();
    printList(L);
}
```

```
int main() {  
    test01();  
  
    system("pause");  
  
    return 0;  
}
```

总结：

- 尾插 --- push_back
- 尾删 --- pop_back
- 头插 --- push_front
- 头删 --- pop_front
- 插入 --- insert
- 删除 --- erase
- 移除 --- remove
- 清空 --- clear

3.7.6 list 数据存取

功能描述：

- 对list容器中数据进行存取

函数原型：

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

list本质上一个链表，不是连续的空间储存数据，所以不支持随机访问（不支持索引）迭代器支持++ -- 但不支持 +1 -1 这种因为这种属于随机访问

示例：

```

#include <list>

//数据存取
void test01()
{
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    //cout << L1.at(0) << endl;//错误 不支持at访问数据
    //cout << L1[0] << endl; //错误 不支持[]方式访问数据
    cout << "第一个元素为: " << L1.front() << endl;
    cout << "最后一个元素为: " << L1.back() << endl;

    //list容器的迭代器是双向迭代器，不支持随机访问
    list<int>::iterator it = L1.begin();
    //it = it + 1;//错误，不可以跳跃访问，即使是+1
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- list容器中不可以通过[]或者at方式访问数据
- 返回第一个元素 --- front
- 返回最后一个元素 --- back

3.7.7 list 反转和排序

功能描述：

- 将容器中的元素反转，以及将容器中的数据进行排序

函数原型：

- reverse(); //反转链表
- sort(); //链表排序

示例：

```

void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

bool myCompare(int val1, int val2)
{
    return val1 > val2;
}

//反转和排序
void test01()
{
    list<int> L;
    L.push_back(90);
    L.push_back(30);
    L.push_back(20);
    L.push_back(70);
    printList(L);

    //反转容器的元素
    L.reverse();
    printList(L);

    //排序
    L.sort(); //默认的排序规则 从小到大
    printList(L);

    L.sort(myCompare); //指定规则，从大到小
    printList(L);
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：

- 反转 --- reverse
- 排序 --- sort （成员函数）

3.7.8 排序案例

案例描述：将Person自定义数据类型进行排序，Person中属性有姓名、年龄、身高

排序规则：按照年龄进行升序，如果年龄相同按照身高进行降序（注意sort函数的成员函数的实现怎么写）

示例：

```
#include <list>
#include <string>
class Person {
public:
    Person(string name, int age , int height) {
        m_Name = name;
        m_Age = age;
        m_Height = height;
    }

public:
    string m_Name; //姓名
    int m_Age; //年龄
    int m_Height; //身高
};

bool ComparePerson(Person& p1, Person& p2) {

    if (p1.m_Age == p2.m_Age) {
        return p1.m_Height > p2.m_Height;
    }
    else
    {
        return p1.m_Age < p2.m_Age;
    }
}

void test01() {

    list<Person> L;

    Person p1("刘备", 35 , 175);
    Person p2("曹操", 45 , 180);
    Person p3("孙权", 40 , 170);
    Person p4("赵云", 25 , 190);
    Person p5("张飞", 35 , 160);
    Person p6("关羽", 35 , 200);

    L.push_back(p1);
    L.push_back(p2);
    L.push_back(p3);
    L.push_back(p4);
    L.push_back(p5);
    L.push_back(p6);

    for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
        cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age
            << " 身高: " << it->m_Height << endl;
    }

    cout << "-----" << endl;
    L.sort(ComparePerson); //排序
```

```

    for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
        cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age
        << " 身高: " << it->m_Height << endl;
    }
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 对于自定义数据类型，必须要指定排序规则，否则编译器不知道如何进行排序
- 高级排序只是在排序规则上再进行一次逻辑规则制定，并不复杂

3.8 set/ multiset 容器

3.8.1 set基本概念

简介：

- 特点：所有元素都会在插入时自动被排序

本质：

- set/multiset属于**关联式容器**，底层结构是用**二叉树**实现。

set和multiset区别：

- set不允许容器中有重复的元素
- multiset允许容器中有重复的元素

3.8.2 set构造和赋值

功能描述：创建set容器以及赋值

构造：

- `set<T> st;` //默认构造函数：
- `set(const set &st);` //拷贝构造函数

赋值：

- `set& operator=(const set &st);` //重载等号操作符

插值

只有insert()

示例：

```
#include <set>

void printSet(set<int> & s)
{
    for (set<int>::iterator it = s.begin(); it != s.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

//构造和赋值
void test01()
{
    set<int> s1;

    s1.insert(10);
    s1.insert(30);
    s1.insert(20);
    s1.insert(40);
    printSet(s1);

    //拷贝构造
    set<int>s2(s1);
    printSet(s2);

    //赋值
    set<int>s3;
    s3 = s2;
    printSet(s3);
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：

- set容器插入数据时用insert
- set容器插入数据的数据会自动排序(不允许插入)

3.8.3 set大小和交换

功能描述：

- 统计set容器大小以及交换set容器

函数原型：

- `size();` //返回容器中元素的数目 set容器不允许resize，因为会有重复数据
- `empty();` //判断容器是否为空
- `swap(st);` //交换两个集合容器

示例：

```
#include <set>

void printSet(set<int> & s)
{
    for (set<int>::iterator it = s.begin(); it != s.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

//大小
void test01()
{
    set<int> s1;

    s1.insert(10);
    s1.insert(30);
    s1.insert(20);
    s1.insert(40);

    if (s1.empty())
    {
        cout << "s1为空" << endl;
    }
    else
    {
        cout << "s1不为空" << endl;
        cout << "s1的大小为: " << s1.size() << endl;
    }
}

//交换
void test02()
{
    set<int> s1;

    s1.insert(10);
    s1.insert(30);
    s1.insert(20);
    s1.insert(40);

    set<int> s2;

    s2.insert(100);
    s2.insert(300);
    s2.insert(200);
    s2.insert(400);

    cout << "交换前" << endl;
    printSet(s1);
    printSet(s2);
    cout << endl;
```

```
cout << "交换后" << endl;
s1.swap(s2);
printSet(s1);
printSet(s2);
}

int main() {
    //test01();
    test02();
    system("pause");
    return 0;
}
```

总结：

- 统计大小 --- size
- 判断是否为空 --- empty
- 交换容器 --- swap

3.8.4 set插入和删除

功能描述：

- set容器进行插入数据和删除数据

函数原型：

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

示例：

```

#include <set>

void printSet(set<int> & s)
{
    for (set<int>::iterator it = s.begin(); it != s.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

//插入和删除
void test01()
{
    set<int> s1;
    //插入
    s1.insert(10);
    s1.insert(30);
    s1.insert(20);
    s1.insert(40);
    printSet(s1);

    //删除
    s1.erase(s1.begin());
    printSet(s1);

    s1.erase(30);
    printSet(s1);

    //清空
    //s1.erase(s1.begin(), s1.end());
    s1.clear();
    printSet(s1);
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：

- 插入 --- insert
- 删除 --- erase
- 清空 --- clear

3.8.5 set查找和统计

功能描述：

- 对set容器进行查找数据以及统计数据

函数原型：

- `find(key);` //查找key是否存在,若存在, 返回该键的元素的迭代器; 若不存在, 返回`set.end()`;
- `count(key);` //统计key的元素个数 (对于set容器只返回0或1)

示例：

```
#include <set>

//查找和统计
void test01()
{
    set<int> s1;
    //插入
    s1.insert(10);
    s1.insert(30);
    s1.insert(20);
    s1.insert(40);

    //查找
    set<int>::iterator pos = s1.find(30);

    if (pos != s1.end())
    {
        cout << "找到了元素 : " << *pos << endl;
    }
    else
    {
        cout << "未找到元素" << endl;
    }

    //统计
    int num = s1.count(30);
    cout << "num = " << num << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：

- 查找 --- find (返回的是迭代器)
- 统计 --- count (对于set, 结果为0或者1)

3.8.6 set和multiset区别

学习目标：

- 掌握set和multiset的区别

区别：

- set不可以插入重复数据，而multiset可以
- set插入数据的同时会返回插入结果，表示插入是否成功
- multiset不会检测数据，因此可以插入重复数据

示例：

```

#include <set>

//set和multiset区别
void test01()
{
    set<int> s;
    //set的insert会返回一个对组的数据类型，两个数据为迭代器和bool，下面相当于ret有两个元素，第一个是迭代器
    pair<set<int>::iterator, bool> ret = s.insert(10);
    if (ret.second) {
        cout << "第一次插入成功!" << endl;
    }
    else {
        cout << "第一次插入失败!" << endl;
    }

    ret = s.insert(10);
    if (ret.second) {
        cout << "第二次插入成功!" << endl;
    }
    else {
        cout << "第二次插入失败!" << endl;
    }

    //multiset
    multiset<int> ms;
    ms.insert(10);
    ms.insert(10);

    for (multiset<int>::iterator it = ms.begin(); it != ms.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 如果不允许插入重复数据可以利用set
- 如果需要插入重复数据利用multiset

3.8.7 pair对组创建

功能描述：

- 成对出现的数据，利用对组可以返回两个数据

两种创建方式：

- `pair<type, type> p (value1, value2);`
- `pair<type, type> p = make_pair(value1, value2);`

示例：

```
#include <string>

//对组创建
void test01()
{
    pair<string, int> p(string("Tom"), 20);
    cout << "姓名: " << p.first << " 年龄: " << p.second << endl;

    pair<string, int> p2 = make_pair("Jerry", 10);
    cout << "姓名: " << p2.first << " 年龄: " << p2.second << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：

两种方式都可以创建对组，记住一种即可

3.8.8 set容器排序

学习目标：

- set容器默认排序规则为从小到大，掌握如何改变排序规则

主要技术点：

- 利用仿函数，可以改变排序规则

示例一 set存放内置数据类型

```

#include <set>

class MyCompare
{
public:
    bool operator()(int v1, int v2) {
        return v1 > v2;
    }
};

void test01()
{
    set<int> s1;
    s1.insert(10);
    s1.insert(40);
    s1.insert(20);
    s1.insert(30);
    s1.insert(50);

    //默认从小到大
    for (set<int>::iterator it = s1.begin(); it != s1.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;

    //指定排序规则
    set<int, MyCompare> s2;
    s2.insert(10);
    s2.insert(40);
    s2.insert(20);
    s2.insert(30);
    s2.insert(50);

    for (set<int, MyCompare>::iterator it = s2.begin(); it != s2.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

总结：利用仿函数可以指定set容器的排序规则

示例二 set存放自定义数据类型

```
#include <set>
#include <string>

class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }

    string m_Name;
    int m_Age;
};

class comparePerson
{
public:
    bool operator()(const Person& p1, const Person &p2)
    {
        //按照年龄进行排序 降序
        return p1.m_Age > p2.m_Age;
    }
};

void test01()
{
    set<Person, comparePerson> s;

    Person p1("刘备", 23);
    Person p2("关羽", 27);
    Person p3("张飞", 25);
    Person p4("赵云", 21);

    s.insert(p1);
    s.insert(p2);
    s.insert(p3);
    s.insert(p4);

    for (set<Person, comparePerson>::iterator it = s.begin(); it != s.end(); it++)
    {
        cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age << endl;
    }
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结：

对于自定义数据类型，set必须指定排序规则才可以插入数据。不然set不知道怎么进行排序

3.9 map/ multimap容器

3.9.1 map基本概念

简介：

- map中所有元素都是pair（对组）
- 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）
- 所有元素都会根据元素的键值自动排序

本质：

- map/multimap属于**关联式容器**，底层结构是用二叉树实现。

优点：

- 可以根据key值快速找到value值

map和multimap区别：

- map不允许容器中有重复key值元素
- multimap允许容器中有重复key值元素

3.9.2 map构造和赋值

功能描述：

- 对map容器进行构造和赋值操作

函数原型：

构造：

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

赋值：

- `map& operator=(const map &mp); //重载等号操作符`

示例：

```

#include <map>

void printMap(map<int, int>&m)
{
    for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
    {
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
    cout << endl;
}

void test01()
{
    map<int, int>m; //默认构造
    m.insert(pair<int, int>(1, 10)); //注意map插入值的方式
    m.insert(pair<int, int>(2, 20));
    m.insert(pair<int, int>(3, 30));
    printMap(m);

    map<int, int>m2(m); //拷贝构造
    printMap(m2);

    map<int, int>m3;
    m3 = m2; //赋值
    printMap(m3);
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：map中所有元素都是成对出现，插入数据时候要使用对组

3.9.3 map大小和交换

功能描述：

- 统计map容器大小以及交换map容器

函数原型：

- size(); //返回容器中元素的数目
- empty(); //判断容器是否为空
- swap(st); //交换两个集合容器

示例：

```

#include <map>

void printMap(map<int, int>&m)
{
    for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
    {
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
    cout << endl;
}

void test01()
{
    map<int, int>m;
    m.insert(pair<int, int>(1, 10));
    m.insert(pair<int, int>(2, 20));
    m.insert(pair<int, int>(3, 30));

    if (m.empty())
    {
        cout << "m为空" << endl;
    }
    else
    {
        cout << "m不为空" << endl;
        cout << "m的大小为: " << m.size() << endl;
    }
}

//交换
void test02()
{
    map<int, int>m;
    m.insert(pair<int, int>(1, 10));
    m.insert(pair<int, int>(2, 20));
    m.insert(pair<int, int>(3, 30));

    map<int, int>m2;
    m2.insert(pair<int, int>(4, 100));
    m2.insert(pair<int, int>(5, 200));
    m2.insert(pair<int, int>(6, 300));

    cout << "交换前" << endl;
    printMap(m);
    printMap(m2);

    cout << "交换后" << endl;
    m.swap(m2);
    printMap(m);
    printMap(m2);
}

int main() {

```

```
test01();  
test02();  
system("pause");  
return 0;  
}
```

总结：

- 统计大小 --- size
- 判断是否为空 --- empty
- 交换容器 --- swap

3.9.4 map插入和删除

功能描述：

- map容器进行插入数据和删除数据

函数原型：

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

示例：

```

#include <map>

void printMap(map<int, int>&m)
{
    for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
    {
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
    cout << endl;
}

void test01()
{
    //插入
    map<int, int> m;
    //第一种插入方式
    m.insert(pair<int, int>(1, 10));
    //第二种插入方式
    m.insert(make_pair(2, 20));
    //第三种插入方式
    m.insert(map<int, int>::value_type(3, 30));
    //第四种插入方式
    m[4] = 40;      //不建议插入，[]可以利用key访问value
    printMap(m);

    //删除
    m.erase(m.begin());
    printMap(m);

    m.erase(3);
    printMap(m);

    //清空
    m.erase(m.begin(), m.end());
    m.clear();
    printMap(m);
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

总结：

- map插入方式很多，记住其一即可，第一种第二中即可
- 插入 --- insert
- 删除 --- erase

- 清空 --- clear

3.9.5 map查找和统计

功能描述：

- 对map容器进行查找数据以及统计数据

函数原型：

- `find(key);` //查找key是否存在,若存在, 返回该键的元素的迭代器; 若不存在, 返回set.end();
- `count(key);` //统计key的元素个数

示例：

```
#include <map>

//查找和统计
void test01()
{
    map<int, int>m;
    m.insert(pair<int, int>(1, 10));
    m.insert(pair<int, int>(2, 20));
    m.insert(pair<int, int>(3, 30));

    //查找
    map<int, int>::iterator pos = m.find(3); //find返回的是迭代器

    if (pos != m.end())
    {
        cout << "找到了元素 key = " << (*pos).first << " value = " << (*pos).second << endl;
    }
    else
    {
        cout << "未找到元素" << endl;
    }

    //统计
    int num = m.count(3); // map只能返回0或1, multimap结果可能大于1.
    cout << "num = " << num << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结：

- 查找 --- find (返回的是迭代器)
- 统计 --- count (对于map, 结果为0或者1)

3.9.6 map容器排序

学习目标:

- map容器默认排序规则为 按照key值进行 从小到大排序, 掌握如何改变排序规则

主要技术点:

- 利用仿函数, 可以改变排序规则

示例:

```
#include <map>

class MyCompare {
public:
    bool operator()(int v1, int v2) {
        return v1 > v2;
    }
};

void test01()
{
    //默认从小到大排序
    //利用仿函数实现从大到小排序
    map<int, int, MyCompare> m;

    m.insert(make_pair(1, 10));
    m.insert(make_pair(2, 20));
    m.insert(make_pair(3, 30));
    m.insert(make_pair(4, 40));
    m.insert(make_pair(5, 50));

    for (map<int, int, MyCompare>::iterator it = m.begin(); it != m.end(); it++) {
        cout << "key:" << it->first << " value:" << it->second << endl;
    }
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

总结:

- 利用仿函数可以指定map容器的排序规则

- 对于自定义数据类型，map必须要指定排序规则,同set容器

3.10 案例-员工分组

3.10.1 案例描述

- 公司今天招聘了10个员工（ABCDEFGHIJ），10名员工进入公司之后，需要指派员工在那个部门工作
- 员工信息有：姓名 工资组成； 部门分为：策划、美术、研发
- 随机给10名员工分配部门和工资
- 通过multimap进行信息的插入 key(部门编号) value(员工)
- 分部门显示员工信息

3.10.2 实现步骤

1. 创建10名员工，放到vector中
2. 遍历vector容器，取出每个员工，进行随机分组
3. 分组后，将员工部门编号作为key，具体员工作为value，放入到multimap容器中
4. 分部门显示员工信息

案例代码：

```

#include<iostream>
using namespace std;
#include <vector>
#include <string>
#include <map>
#include <ctime>

/*
- 公司今天招聘了10个员工 (ABCDEFGHIJ) , 10名员工进入公司之后, 需要指派员工在那个部门工作
- 员工信息有: 姓名 工资组成; 部门分为: 策划、美术、研发
- 随机给10名员工分配部门和工资
- 通过multimap进行信息的插入 key(部门编号) value(员工)
- 分部门显示员工信息
*/
#define CEHUA 0
#define MEISHU 1
#define YANFA 2

class Worker
{
public:
    string m_Name;
    int m_Salary;
};

void createWorker(vector<Worker>&v)
{
    string nameSeed = "ABCDEFGHIJ";
    for (int i = 0; i < 10; i++)
    {
        Worker worker;
        worker.m_Name = "员工";
        worker.m_Name += nameSeed[i];

        worker.m_Salary = rand() % 10000 + 10000; // 10000 ~ 19999
        //将员工放入到容器中
        v.push_back(worker);
    }
}

//员工分组
void setGroup(vector<Worker>&v, multimap<int, Worker>&m)
{
    for (vector<Worker>::iterator it = v.begin(); it != v.end(); it++)
    {
        //产生随机部门编号
        int deptId = rand() % 3; // 0 1 2

        //将员工插入到分组中
        //key部门编号, value具体员工
        m.insert(make_pair(deptId, *it));
    }
}

```

```

void showWorkerByGourp(multimap<int, Worker>&m)
{
    // 0 A B C 1 D E 2 F G ...
    cout << "策划部门: " << endl;

    multimap<int, Worker>::iterator pos = m.find(CEHUA);
    int count = m.count(CEHUA); // 统计具体人数
    int index = 0;
    for (; pos != m.end() && index < count; pos++, index++)
    {
        cout << "姓名: " << pos->second.m_Name << " 工资: " << pos->second.m_Salary <<
    }

    cout << "-----" << endl;
    cout << "美术部门: " << endl;
    pos = m.find(MEISHU);
    count = m.count(MEISHU); // 统计具体人数
    index = 0;
    for (; pos != m.end() && index < count; pos++, index++)
    {
        cout << "姓名: " << pos->second.m_Name << " 工资: " << pos->second.m_Salary <<
    }

    cout << "-----" << endl;
    cout << "研发部门: " << endl;
    pos = m.find(YANFA);
    count = m.count(YANFA); // 统计具体人数
    index = 0;
    for (; pos != m.end() && index < count; pos++, index++)
    {
        cout << "姓名: " << pos->second.m_Name << " 工资: " << pos->second.m_Salary <<
    }
}

int main() {
    srand((unsigned int)time(NULL));

    //1、创建员工
    vector<Worker> vWorker;
    createWorker(vWorker);

    //2、员工分组
    multimap<int, Worker> mWorker;
    setGroup(vWorker, mWorker);

    //3、分组显示员工
    showWorkerByGourp(mWorker);

    ////测试
    //for (vector<Worker>::iterator it = vWorker.begin(); it != vWorker.end(); it++)
    //{
    //    cout << "姓名: " << it->m_Name << " 工资: " << it->m_Salary << endl;
}

```

```
//}  
system("pause");  
return 0;  
}
```

总结：

- 当数据以键值对形式存在，可以考虑用map 或 multimap

4 STL- 函数对象

4.1 函数对象（仿函数）

4.1.1 函数对象概念

概念：

- 重载函数调用操作符的类，其对象常称为**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(仿函数)是一个**类**，不是一个函数

4.1.2 函数对象使用

特点：

- 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

示例：

```
#include <string>

//1、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
class MyAdd
{
public :
    int operator()(int v1,int v2)
    {
        return v1 + v2;
    }
};

void test01()
{
    MyAdd myAdd;
    cout << myAdd(10, 10) << endl;
}

//2、函数对象可以有自己的状态
class MyPrint
{
public:
    MyPrint()
    {
        count = 0;
    }
    void operator()(string test)
    {
        cout << test << endl;
        count++; //统计使用次数
    }

    int count; //内部自己的状态
};

void test02()
{
    MyPrint myPrint;
    myPrint("hello world");
    myPrint("hello world");
    myPrint("hello world");
    cout << "myPrint调用次数为: " << myPrint.count << endl;
}

//3、函数对象可以作为参数传递
void doPrint(MyPrint &mp , string test)
{
    mp(test);
}

void test03()
{
    MyPrint myPrint;
    doPrint(myPrint, "Hello C++");
}
```

```
int main() {  
    //test01();  
    //test02();  
    test03();  
  
    system("pause");  
  
    return 0;  
}
```

总结：

- 仿函数写法非常灵活，可以作为参数进行传递。

4.2 谓词

4.2.1 谓词概念

概念：

- 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

4.2.2 一元谓词

示例：

```

#include <vector>
#include <algorithm>

//1.一元谓词
struct GreaterFive{
    bool operator()(int val) {
        return val > 5;
    }
};

void test01() {

    vector<int> v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    //GreaterFive()匿名的函数对象
    vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterFive());
    if (it == v.end()) {
        cout << "没找到!" << endl;
    }
    else {
        cout << "找到:" << *it << endl;
    }
}

int main() {

    test01();
    system("pause");
    return 0;
}

```

总结：参数只有一个的谓词，称为一元谓词

4.2.3 二元谓词

示例：

```

#include <vector>
#include <algorithm>
//二元谓词
class MyCompare
{
public:
    bool operator()(int num1, int num2)
    {
        return num1 > num2;
    }
};

void test01()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(40);
    v.push_back(20);
    v.push_back(30);
    v.push_back(50);

    //默认从小到大
    sort(v.begin(), v.end());
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
    cout << "-----" << endl;

    //使用函数对象改变算法策略，排序从大到小
    sort(v.begin(), v.end(), MyCompare());
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：参数只有两个的谓词，称为二元谓词

4.3 内建函数对象

4.3.1 内建函数对象意义

概念：

- STL内建了一些函数对象

分类：

- 算术仿函数
- 关系仿函数
- 逻辑仿函数

用法：

- 这些仿函数所产生的对象，用法和一般函数完全相同
- 使用内建函数对象，需要引入头文件 `#include<functional>`

4.3.2 算术仿函数

功能描述：

- 实现四则运算
- 其中`negate`是一元运算，其他都是二元运算

仿函数原型：

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

示例：

```

#include <functional>
//negate
void test01()
{
    negate<int> n;
    cout << n(50) << endl;
}

//plus
void test02()
{
    plus<int> p;
    cout << p(10, 20) << endl;
}

int main() {
    test01();
    test02();

    system("pause");
    return 0;
}

```

总结：使用内建函数对象时，需要引入头文件 `#include <functional>`

4.3.3 关系仿函数

功能描述：

- 实现关系对比

仿函数原型：

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

示例：

```

#include <functional>
#include <vector>
#include <algorithm>

class MyCompare
{
public:
    bool operator()(int v1, int v2)
    {
        return v1 > v2;
    }
};

void test01()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(30);
    v.push_back(50);
    v.push_back(40);
    v.push_back(20);

    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;

    //自己实现仿函数
    //sort(v.begin(), v.end(), MyCompare());
    //STL内建仿函数 大于仿函数
    sort(v.begin(), v.end(), greater<int>());

    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

总结：关系仿函数中最常用的就是greater<>大于

4.3.4 逻辑仿函数

功能描述：

- 实现逻辑运算

函数原型：

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

示例：

```
#include <vector>
#include <functional>
#include <algorithm>
void test01()
{
    vector<bool> v;
    v.push_back(true);
    v.push_back(false);
    v.push_back(true);
    v.push_back(false);

    for (vector<bool>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    //逻辑非 将v容器搬运到v2中，并执行逻辑非运算
    vector<bool> v2;
    v2.resize(v.size());
    transform(v.begin(), v.end(), v2.begin(), logical_not<bool>());
    for (vector<bool>::iterator it = v2.begin(); it != v2.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：逻辑仿函数实际应用较少，了解即可

5 STL- 常用算法

概述:

- 算法主要是由头文件 `<algorithm>` `<functional>` `<numeric>` 组成。
- `<algorithm>` 是所有STL头文件中最大的一个，范围涉及到比较、交换、查找、遍历操作、复制、修改等
- `<numeric>` 体积很小，只包括几个在序列上面进行简单数学运算的模板函数
- `<functional>` 定义了一些模板类,用以声明函数对象。

5.1 常用遍历算法

学习目标:

- 掌握常用的遍历算法

算法简介:

- `for_each` //遍历容器
- `transform` //搬运容器到另一个容器中

5.1.1 for_each

功能描述:

- 实现遍历容器

函数原型:

- `for_each(iterator beg, iterator end, _func);`
// 遍历算法 遍历容器元素
// beg 开始迭代器
// end 结束迭代器
// _func 函数或者函数对象

示例:

```

#include <algorithm>
#include <vector>

//普通函数
void print01(int val)
{
    cout << val << " ";
}

//函数对象
class print02
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

//for_each算法基本用法
void test01() {

    vector<int> v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }

    //遍历算法
    for_each(v.begin(), v.end(), print01);
    cout << endl;

    for_each(v.begin(), v.end(), print02());
    cout << endl;
}

int main() {

    test01();

    system("pause");
    return 0;
}

```

总结：for_each在实际开发中是最常用遍历算法，需要熟练掌握

5.1.2 transform

功能描述：

- 搬运容器到另一个容器中

函数原型：

- `transform(iterator beg1, iterator end1, iterator beg2, _func);`

//beg1 源容器开始迭代器

//end1 源容器结束迭代器

//beg2 目标容器开始迭代器

//_func 函数或者函数对象

示例：

```

#include<vector>
#include<algorithm>

//常用遍历算法 搬运 transform

class TransForm
{
public:
    int operator()(int val)
    {
        return val;
    }
};

class MyPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int>v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }

    vector<int>vTarget; //目标容器

    vTarget.resize(v.size()); // 目标容器需要提前开辟空间

    transform(v.begin(), v.end(), vTarget.begin(), TransForm());

    for_each(vTarget.begin(), vTarget.end(), MyPrint());
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

总结：搬运的目标容器必须要提前开辟空间，否则无法正常搬运

5.2 常用查找算法

学习目标：

- 掌握常用的查找算法

算法简介：

- `find` //查找元素
- `find_if` //按条件查找元素
- `adjacent_find` //查找相邻重复元素
- `binary_search` //二分查找法
- `count` //统计元素个数
- `count_if` //按条件统计元素个数

5.2.1 find

功能描述：

- 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`end()`

函数原型：

- `find(iterator beg, iterator end, value);`
// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
// beg 开始迭代器
// end 结束迭代器
// value 查找的元素

示例：

```
#include <algorithm>
#include <vector>
#include <string>
void test01() {

    vector<int> v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i + 1);
    }
    //查找容器中是否有 5 这个元素
    vector<int>::iterator it = find(v.begin(), v.end(), 5);
    if (it == v.end())
    {
        cout << "没有找到!" << endl;
    }
    else
    {
        cout << "找到:" << *it << endl;
    }
}

class Person {
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
    //重载==
    bool operator==(const Person& p)
    {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
        {
            return true;
        }
        return false;
    }

public:
    string m_Name;
    int m_Age;
};

void test02() {

    vector<Person> v;

    //创建数据
    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);

    v.push_back(p1);
    v.push_back(p2);
```

```
v.push_back(p3);
v.push_back(p4);

vector<Person>::iterator it = find(v.begin(), v.end(), p2);
if (it == v.end())
{
    cout << "没有找到!" << endl;
}
else
{
    cout << "找到姓名:" << it->m_Name << " 年龄: " << it->m_Age << endl;
}
}
```

总结：利用find可以在容器中找指定的元素，返回值是**迭代器**

5.2.2 find_if

功能描述：

- 按条件查找元素

函数原型：

- `find_if(iterator beg, iterator end, _Pred);`
// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
// beg 开始迭代器
// end 结束迭代器
// _Pred 函数或者谓词（返回bool类型的仿函数）

示例：

```
#include <algorithm>
#include <vector>
#include <string>

//内置数据类型
class GreaterFive
{
public:
    bool operator()(int val)
    {
        return val > 5;
    }
};

void test01() {

    vector<int> v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i + 1);
    }

    vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterFive());
    if (it == v.end()) {
        cout << "没有找到!" << endl;
    }
    else {
        cout << "找到大于5的数字:" << *it << endl;
    }
}

//自定义数据类型
class Person {
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
public:
    string m_Name;
    int m_Age;
};

class Greater20
{
public:
    bool operator()(Person &p)
    {
        return p.m_Age > 20;
    }
};

void test02() {
```

```

vector<Person> v;

//创建数据
Person p1("aaa", 10);
Person p2("bbb", 20);
Person p3("ccc", 30);
Person p4("ddd", 40);

v.push_back(p1);
v.push_back(p2);
v.push_back(p3);
v.push_back(p4);

vector<Person>::iterator it = find_if(v.begin(), v.end(), Greater20());
if (it == v.end())
{
    cout << "没有找到!" << endl;
}
else
{
    cout << "找到姓名:" << it->m_Name << " 年龄: " << it->m_Age << endl;
}
}

int main() {

    //test01();

    test02();

    system("pause");

    return 0;
}

```

总结：find_if按条件查找使查找更加灵活，提供的仿函数可以改变不同的策略

5.2.3 adjacent_find

功能描述：

- 查找相邻重复元素

函数原型：

- adjacent_find(iterator beg, iterator end);
 // 查找相邻重复元素,返回相邻元素的第一个位置的迭代器
 // beg 开始迭代器
 // end 结束迭代器

示例：

```

#include <algorithm>
#include <vector>

void test01()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(5);
    v.push_back(2);
    v.push_back(4);
    v.push_back(4);
    v.push_back(3);

    //查找相邻重复元素
    vector<int>::iterator it = adjacent_find(v.begin(), v.end());
    if (it == v.end()) {
        cout << "找不到!" << endl;
    }
    else {
        cout << "找到相邻重复元素为:" << *it << endl;
    }
}

```

总结：面试题中如果出现查找相邻重复元素，记得用STL中的adjacent_find算法

5.2.4 binary_search

功能描述：

- 查找指定元素是否存在

函数原型：

- bool binary_search(iterator beg, iterator end, value);

// 查找指定的元素，查到 返回true 否则false

// 注意：在**无序序列中不可用**

// beg 开始迭代器

// end 结束迭代器

// value 查找的元素

示例：

```

#include <algorithm>
#include <vector>

void test01()
{
    vector<int>v;

    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    //二分查找
    bool ret = binary_search(v.begin(), v.end(), 2);
    if (ret)
    {
        cout << "找到了" << endl;
    }
    else
    {
        cout << "未找到" << endl;
    }
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：二分查找法查找效率很高，值得注意的是查找的容器中元素必须的有序序列

5.2.5 count

功能描述：

- 统计元素个数

函数原型：

- count(iterator beg, iterator end, value);
 // 统计元素出现次数
 // beg 开始迭代器
 // end 结束迭代器
 // value 统计的元素

示例：

```
#include <algorithm>
#include <vector>

//内置数据类型
void test01()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(4);
    v.push_back(5);
    v.push_back(3);
    v.push_back(4);
    v.push_back(4);

    int num = count(v.begin(), v.end(), 4);

    cout << "4的个数为: " << num << endl;
}

//自定义数据类型
class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
    bool operator==(const Person & p)
    {
        if (this->m_Age == p.m_Age)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    string m_Name;
    int m_Age;
};

void test02()
{
    vector<Person> v;

    Person p1("刘备", 35);
    Person p2("关羽", 35);
    Person p3("张飞", 35);
    Person p4("赵云", 30);
    Person p5("曹操", 25);

    v.push_back(p1);
```

```
v.push_back(p2);
v.push_back(p3);
v.push_back(p4);
v.push_back(p5);

Person p("诸葛亮", 35);

int num = count(v.begin(), v.end(), p);
cout << "num = " << num << endl;
}

int main() {

    //test01();

    test02();

    system("pause");

    return 0;
}
```

总结：统计自定义数据类型时候，需要配合重载 operator==

5.2.6 count_if

功能描述：

- 按条件统计元素个数

函数原型：

- `count_if(iterator beg, iterator end, _Pred);`
// 按条件统计元素出现次数
// beg 开始迭代器
// end 结束迭代器
// _Pred 谓词

示例：

```
#include <algorithm>
#include <vector>

class Greater4
{
public:
    bool operator()(int val)
    {
        return val >= 4;
    }
};

//内置数据类型
void test01()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(4);
    v.push_back(5);
    v.push_back(3);
    v.push_back(4);
    v.push_back(4);

    int num = count_if(v.begin(), v.end(), Greater4());

    cout << "大于4的个数为: " << num << endl;
}

//自定义数据类型
class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }

    string m_Name;
    int m_Age;
};

class AgeLess35
{
public:
    bool operator()(const Person &p)
    {
        return p.m_Age < 35;
    }
};

void test02()
{
    vector<Person> v;
```

```

Person p1("刘备", 35);
Person p2("关羽", 35);
Person p3("张飞", 35);
Person p4("赵云", 30);
Person p5("曹操", 25);

v.push_back(p1);
v.push_back(p2);
v.push_back(p3);
v.push_back(p4);
v.push_back(p5);

int num = count_if(v.begin(), v.end(), AgeLess35());
cout << "小于35岁的个数: " << num << endl;
}

int main() {
    //test01();

    test02();

    system("pause");
    return 0;
}

```

总结：按值统计用count，按条件统计用count_if

5.3 常用排序算法

学习目标：

- 掌握常用的排序算法

算法简介：

- sort //对容器内元素进行排序
- random_shuffle //洗牌 指定范围内的元素随机调整次序
- merge // 容器元素合并，并存储到另一容器中
- reverse // 反转指定范围的元素

5.3.1 sort

功能描述：

- 对容器内元素进行排序

函数原型：

- `sort(iterator beg, iterator end, _Pred);`
 // 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
 // beg 开始迭代器
 // end 结束迭代器
 // _Pred 谓词

示例：

```
#include <algorithm>
#include <vector>

void myPrint(int val)
{
    cout << val << " ";
}

void test01() {
    vector<int> v;
    v.push_back(10);
    v.push_back(30);
    v.push_back(50);
    v.push_back(20);
    v.push_back(40);

    //sort默认从小到大排序
    sort(v.begin(), v.end());
    for_each(v.begin(), v.end(), myPrint);
    cout << endl;

    //从大到小排序
    sort(v.begin(), v.end(), greater<int>());
    for_each(v.begin(), v.end(), myPrint);
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：sort属于开发中最常用的算法之一，需熟练掌握

5.3.2 random_shuffle

功能描述：

- 洗牌 指定范围内的元素随机调整次序

函数原型：

- random_shuffle(iterator beg, iterator end);
// 指定范围内的元素随机调整次序
// beg 开始迭代器
// end 结束迭代器

示例：

```
#include <algorithm>
#include <vector>
#include <ctime>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    srand((unsigned int)time(NULL));
    vector<int> v;
    for(int i = 0 ; i < 10;i++)
    {
        v.push_back(i);
    }
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;

    //打乱顺序
    random_shuffle(v.begin(), v.end());
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：random_shuffle洗牌算法比较实用，使用时记得加随机数种子

5.3.3 merge

功能描述:

- 两个容器元素合并，并存储到另一容器中

函数原型:

```
• merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);  
// 容器元素合并，并存储到另一容器中  
// 注意: 两个容器必须是有序的  
// beg1 容器1开始迭代器  
// end1 容器1结束迭代器  
// beg2 容器2开始迭代器  
// end2 容器2结束迭代器  
// dest 目标容器开始迭代器
```

示例:

```

#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10 ; i++)
    {
        v1.push_back(i);
        v2.push_back(i + 1);
    }

    vector<int> vttarget;
    //目标容器需要提前开辟空间
    vttarget.resize(v1.size() + v2.size());
    //合并 需要两个有序序列
    merge(v1.begin(), v1.end(), v2.begin(), v2.end(), vttarget.begin());
    for_each(vttarget.begin(), vttarget.end(), myPrint());
    cout << endl;
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：merge合并的两个容器必须的有序序列

5.3.4 reverse

功能描述：

- 将容器内元素进行反转

函数原型：

- reverse(iterator beg, iterator end);
// 反转指定范围的元素

```
// beg 开始迭代器  
// end 结束迭代器
```

示例：

```
#include <algorithm>  
#include <vector>  
  
class myPrint  
{  
public:  
    void operator()(int val)  
    {  
        cout << val << " ";  
    }  
};  
  
void test01()  
{  
    vector<int> v;  
    v.push_back(10);  
    v.push_back(30);  
    v.push_back(50);  
    v.push_back(20);  
    v.push_back(40);  
  
    cout << "反转前: " << endl;  
    for_each(v.begin(), v.end(), myPrint());  
    cout << endl;  
  
    cout << "反转后: " << endl;  
  
    reverse(v.begin(), v.end());  
    for_each(v.begin(), v.end(), myPrint());  
    cout << endl;  
}  
  
int main() {  
    test01();  
  
    system("pause");  
  
    return 0;  
}
```

总结：reverse反转区间内元素，面试题可能涉及到

5.4 常用拷贝和替换算法

学习目标：

- 掌握常用的拷贝和替换算法

算法简介：

- `copy` // 容器内指定范围的元素拷贝到另一容器中
- `replace` // 将容器内指定范围的旧元素修改为新元素
- `replace_if` // 容器内指定范围满足条件的元素替换为新元素
- `swap` // 互换两个容器的元素

5.4.1 copy

功能描述：

- 容器内指定范围的元素拷贝到另一容器中

函数原型：

- `copy(iterator beg, iterator end, iterator dest);`
// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
// beg 开始迭代器
// end 结束迭代器
// dest 目标起始迭代器

示例：

```

#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i + 1);
    }
    vector<int> v2;
    v2.resize(v1.size());
    copy(v1.begin(), v1.end(), v2.begin());

    for_each(v2.begin(), v2.end(), myPrint());
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：利用copy算法在拷贝时，目标容器记得提前开辟空间

5.4.2 replace

功能描述：

- 将容器内指定范围的旧元素修改为新元素

函数原型：

- replace(iterator beg, iterator end, oldvalue, newvalue);
 // 将区间内旧元素 替换成 新元素
 // beg 开始迭代器
 // end 结束迭代器
 // oldvalue 旧元素
 // newvalue 新元素

示例：

```
#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v;
    v.push_back(20);
    v.push_back(30);
    v.push_back(20);
    v.push_back(40);
    v.push_back(50);
    v.push_back(10);
    v.push_back(20);

    cout << "替换前：" << endl;
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;

    //将容器中的20 替换成 2000
    cout << "替换后：" << endl;
    replace(v.begin(), v.end(), 20, 2000);
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：replace会替换区间内满足条件的元素

5.4.3 replace_if

功能描述：

- 将区间内满足条件的元素，替换成指定元素

函数原型：

- `replace_if(iterator beg, iterator end, _pred, newvalue);`
// 按条件替换元素，满足条件的替换成指定元素
// beg 开始迭代器
// end 结束迭代器
// _pred 谓词
// newvalue 替换的新元素

示例：

```

#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

class ReplaceGreater30
{
public:
    bool operator()(int val)
    {
        return val >= 30;
    }
};

void test01()
{
    vector<int> v;
    v.push_back(20);
    v.push_back(30);
    v.push_back(20);
    v.push_back(40);
    v.push_back(50);
    v.push_back(10);
    v.push_back(20);

    cout << "替换前: " << endl;
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;

    //将容器中大于等于的30 替换成 3000
    cout << "替换后: " << endl;
    replace_if(v.begin(), v.end(), ReplaceGreater30(), 3000);
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}

```

总结：replace_if按条件查找，可以利用仿函数灵活筛选满足的条件

5.4.4 swap

功能描述：

- 互换两个容器的元素

函数原型：

- `swap(container c1, container c2);`
 // 互换两个容器的元素
 // c1容器1
 // c2容器2

示例：

```

#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i+100);
    }

    cout << "交换前: " << endl;
    for_each(v1.begin(), v1.end(), myPrint());
    cout << endl;
    for_each(v2.begin(), v2.end(), myPrint());
    cout << endl;

    cout << "交换后: " << endl;
    swap(v1, v2);
    for_each(v1.begin(), v1.end(), myPrint());
    cout << endl;
    for_each(v2.begin(), v2.end(), myPrint());
    cout << endl;
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：swap交换容器时，注意交换的容器要同种类型

5.5 常用算术生成算法

学习目标：

- 掌握常用的算术生成算法

注意：

- 算术生成算法属于小型算法，使用时包含的头文件为 `#include <numeric>`

算法简介：

- `accumulate` // 计算容器元素累计总和
- `fill` // 向容器中添加元素

5.5.1 accumulate

功能描述：

- 计算区间内 容器元素累计总和

函数原型：

- `accumulate(iterator beg, iterator end, value);`
// 计算容器元素累计总和
// beg 开始迭代器
// end 结束迭代器
// value 起始值

示例：

```
#include <numeric>
#include <vector>
void test01()
{
    vector<int> v;
    for (int i = 0; i <= 100; i++) {
        v.push_back(i);
    }

    int total = accumulate(v.begin(), v.end(), 0);

    cout << "total = " << total << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

总结：`accumulate`使用时头文件注意是 `numeric`，这个算法很实用

5.5.2 fill

功能描述：

- 向容器中填充指定的元素

函数原型：

- `fill(iterator beg, iterator end, value);`
// 向容器中填充元素
// beg 开始迭代器
// end 结束迭代器
// value 填充的值

示例：

```
#include <numeric>
#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v;
    v.resize(10);
    //填充
    fill(v.begin(), v.end(), 100);

    for_each(v.begin(), v.end(), myPrint());
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}
```

****总结：** **利用fill可以将容器区间内元素填充为 指定的值

5.6 常用集合算法

学习目标：

- 掌握常用的集合算法

算法简介：

- set_intersection // 求两个容器的交集
- set_union // 求两个容器的并集
- set_difference // 求两个容器的差集

5.6.1 set_intersection

功能描述：

- 求两个容器的交集

函数原型：

- set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
// 求两个集合的交集
// 注意：两个集合必须是有序序列
// beg1 容器1开始迭代器
// end1 容器1结束迭代器
// beg2 容器2开始迭代器
// end2 容器2结束迭代器
// dest 目标容器开始迭代器

示例：

```

#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
        v2.push_back(i+5);
    }

    vector<int> vTarget;
    //取两个里面较小的值给目标容器开辟空间
    vTarget.resize(min(v1.size(), v2.size()));

    //返回目标容器的最后一个元素的迭代器地址
    vector<int>::iterator itEnd =
    set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());

    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;
}

int main() {
    test01();
    system("pause");
    return 0;
}

```

总结：

- 求交集的两个集合必须的有序序列
- 目标容器开辟空间需要从**两个容器中取小值**
- set_intersection返回值既是交集中最后一个元素的位置

5.6.2 set_union

功能描述：

- 求两个集合的并集

函数原型：

```
• set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);  
// 求两个集合的并集  
// 注意:两个集合必须是有序序列  
// beg1 容器1开始迭代器  
// end1 容器1结束迭代器  
// beg2 容器2开始迭代器  
// end2 容器2结束迭代器  
// dest 目标容器开始迭代器
```

示例：

```

#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i+5);
    }

    vector<int> vTarget;
    //取两个容器的和给目标容器开辟空间
    vTarget.resize(v1.size() + v2.size());

    //返回目标容器的最后一个元素的迭代器地址
    vector<int>::iterator itEnd =
    set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());

    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 求并集的两个集合必须的有序序列
- 目标容器开辟空间需要**两个容器相加**
- set_union返回值既是并集中最后一个元素的位置

5.6.3 set_difference

功能描述：

- 求两个集合的差集

函数原型:

```
• set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);  
// 求两个集合的差集  
// 注意:两个集合必须是有序序列  
// beg1 容器1开始迭代器  
// end1 容器1结束迭代器  
// beg2 容器2开始迭代器  
// end2 容器2结束迭代器  
// dest 目标容器开始迭代器
```

示例:

```

#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i+5);
    }

    vector<int> vTarget;
    //取两个里面较大的值给目标容器开辟空间
    vTarget.resize( max(v1.size() , v2.size()) );

    //返回目标容器的最后一个元素的迭代器地址
    cout << "v1与v2的差集为: " << endl;
    vector<int>::iterator itEnd =
    set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());
    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;

    cout << "v2与v1的差集为: " << endl;
    itEnd = set_difference(v2.begin(), v2.end(), v1.begin(), v1.end(), vTarget.begin());
    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;
}

int main() {
    test01();

    system("pause");
    return 0;
}

```

总结：

- 求差集的两个集合必须的有序序列
- 目标容器开辟空间需要从**两个容器取较大值**
- set_difference返回值既是差集中最后一个元素的位置

通讯录管理系统

1、系统需求

通讯录是一个可以记录亲人、好友信息的工具。

本教程主要利用C++来实现一个通讯录管理系统

系统中需要实现的功能如下：

- 添加联系人：向通讯录中添加新人，信息包括（姓名、性别、年龄、联系电话、家庭住址）最多记录1000人
- 显示联系人：显示通讯录中所有联系人信息
- 删除联系人：按照姓名进行删除指定联系人
- 查找联系人：按照姓名查看指定联系人信息
- 修改联系人：按照姓名重新修改指定联系人
- 清空联系人：清空通讯录中所有信息
- 退出通讯录：退出当前使用的通讯录

2、创建项目

创建项目步骤如下：

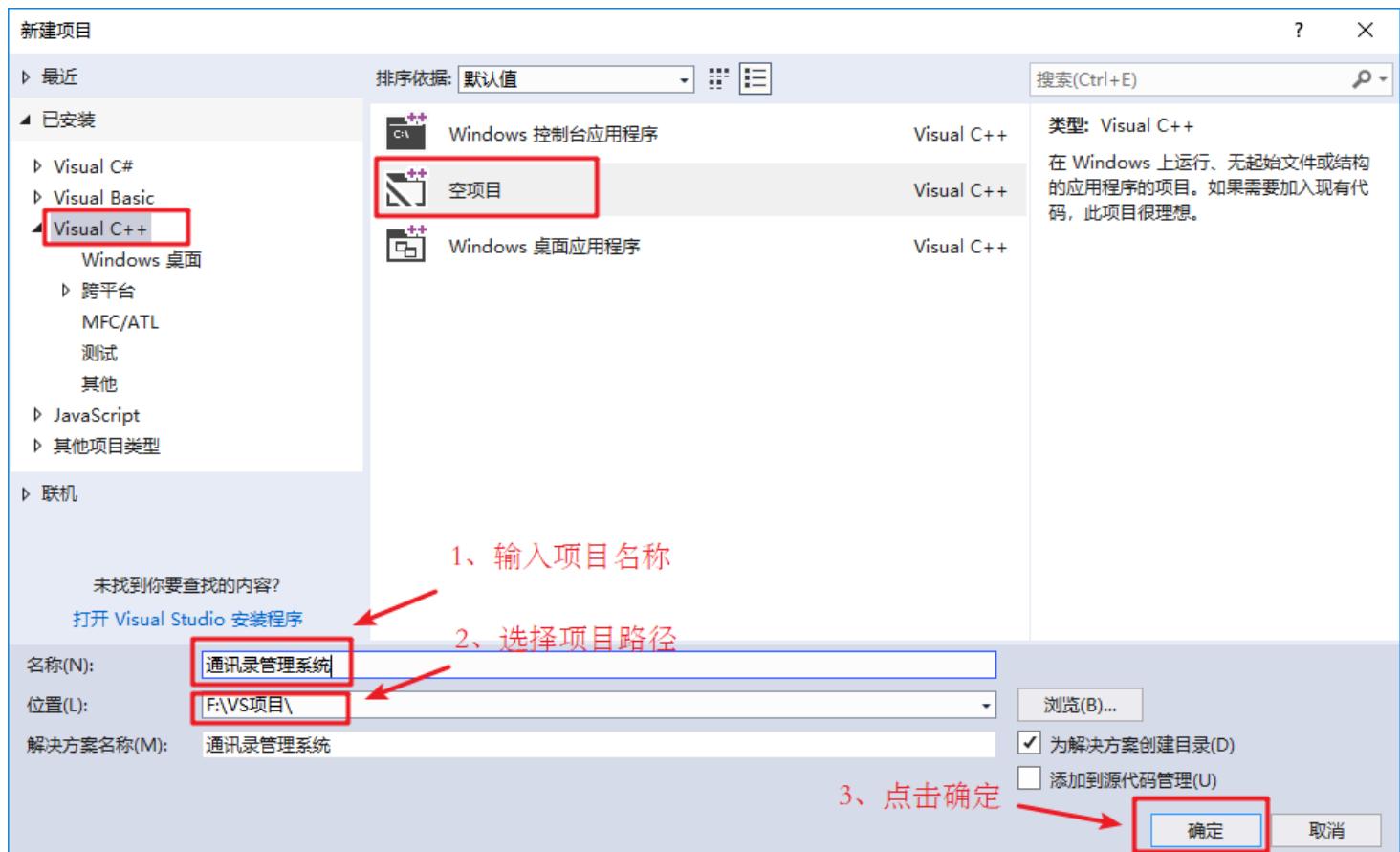
- 创建新项目
- 添加文件

2.1 创建项目

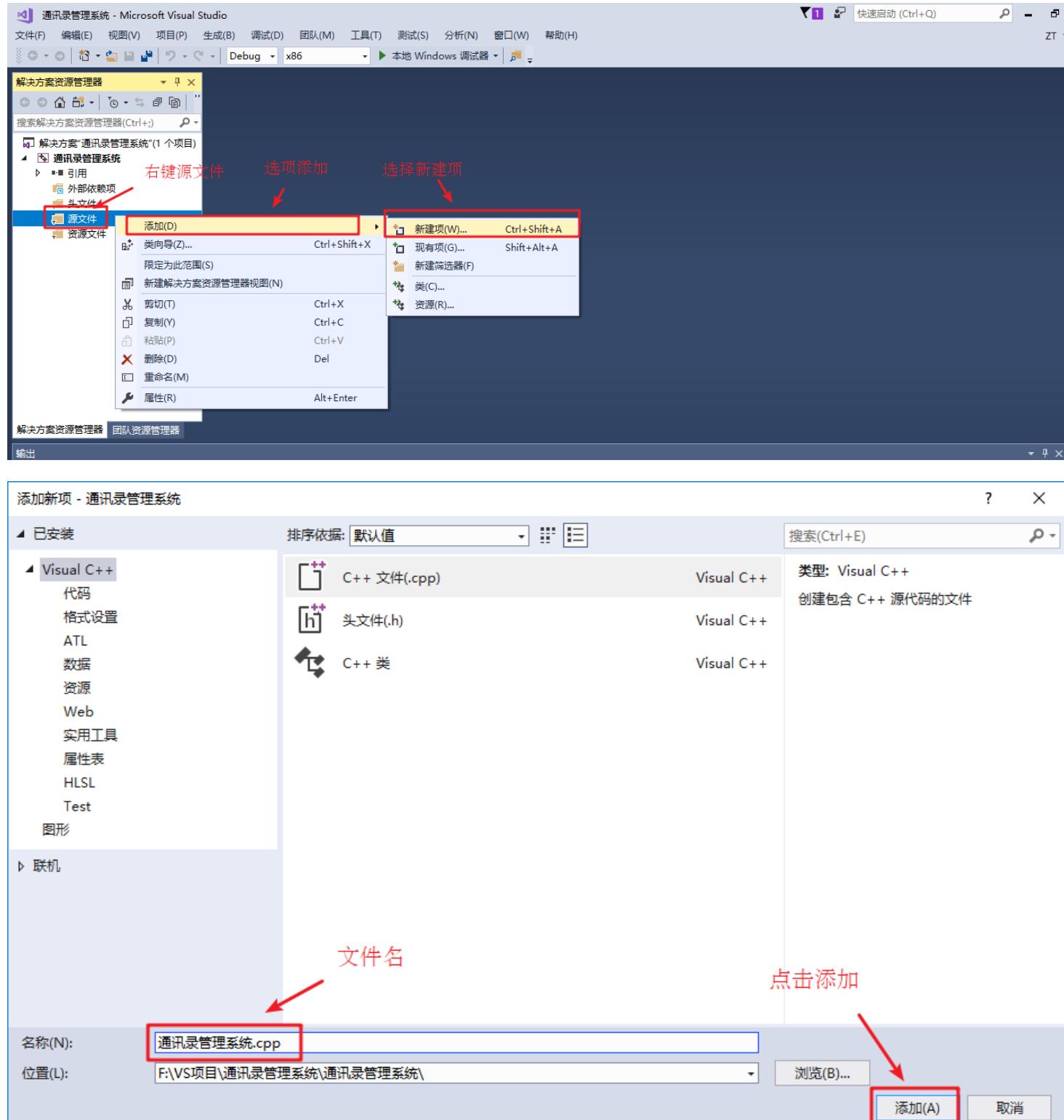
打开vs2017后，点击创建新项目，创建新的C++项目



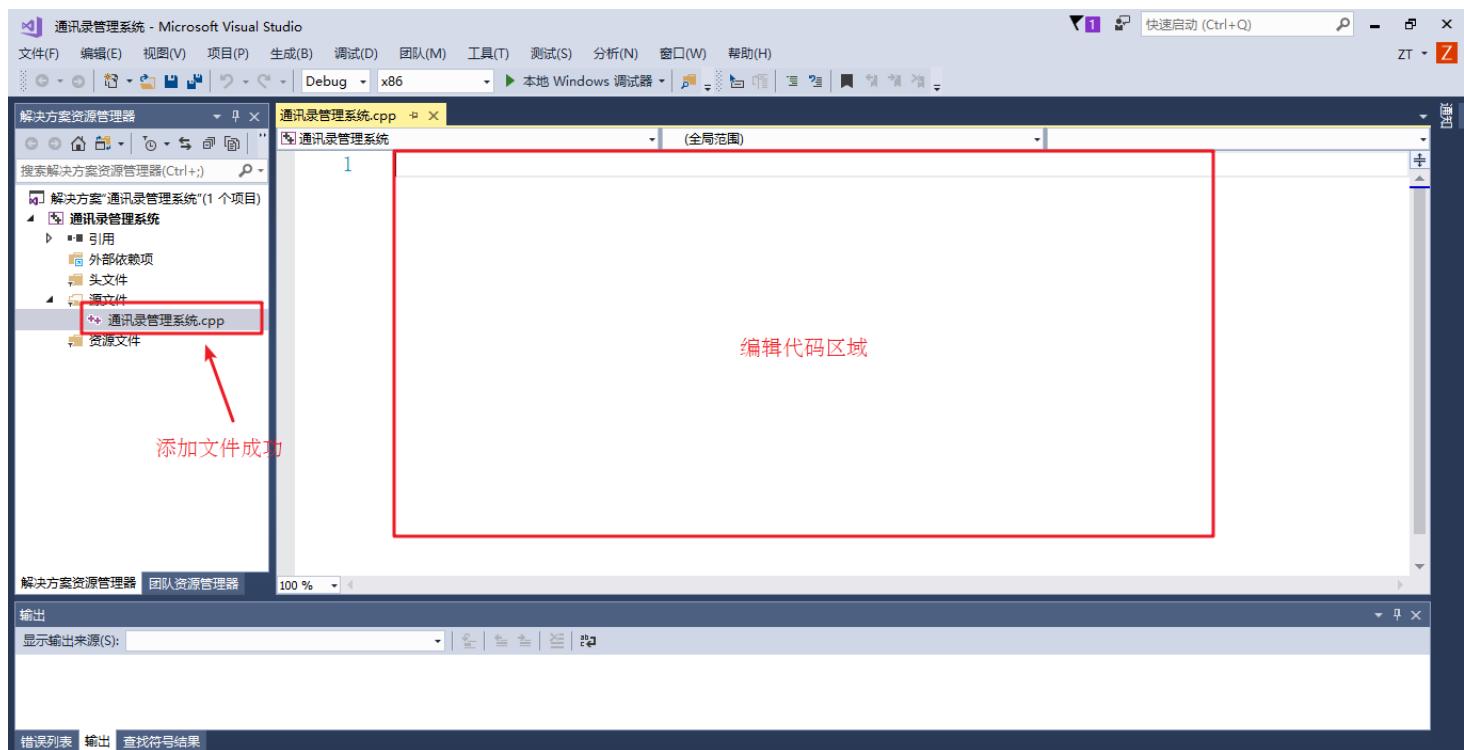
填写项目名称，选择项目路径



2.2添加文件



添加成功后，效果如图：



至此，项目已创建完毕

3、菜单功能

功能描述： 用户选择功能的界面

菜单界面效果如下图：

```
*****  
***** 1、添加联系人 *****  
***** 2、显示联系人 *****  
***** 3、删除联系人 *****  
***** 4、查找联系人 *****  
***** 5、修改联系人 *****  
***** 6、清空联系人 *****  
***** 0、退出通讯录 *****  
*****
```

步骤：

- 封装函数显示该界面 如 void showMenu()
- 在main函数中调用封装好的函数

代码：

```
#include<iostream>
using namespace std;

//菜单界面
void showMenu()
{
    cout << "*****" << endl;
    cout << "***** 1、添加联系人 *****" << endl;
    cout << "***** 2、显示联系人 *****" << endl;
    cout << "***** 3、删除联系人 *****" << endl;
    cout << "***** 4、查找联系人 *****" << endl;
    cout << "***** 5、修改联系人 *****" << endl;
    cout << "***** 6、清空联系人 *****" << endl;
    cout << "***** 0、退出通讯录 *****" << endl;
    cout << "*****" << endl;
}

int main() {
    showMenu();

    system("pause");

    return 0;
}
```

4、退出功能

功能描述：退出通讯录系统

思路：根据用户不同的选择，进入不同的功能，可以选择switch分支结构，将整个架构进行搭建

当用户选择0时候，执行退出，选择其他先不做操作，也不会退出程序

代码：

```
int main() {
    int select = 0;
    while (true)
    {
        showMenu();
        cin >> select;
        switch (select)
        {
            case 1: //添加联系人
                break;
            case 2: //显示联系人
                break;
            case 3: //删除联系人
                break;
            case 4: //查找联系人
                break;
            case 5: //修改联系人
                break;
            case 6: //清空联系人
                break;
            case 0: //退出通讯录
                cout << "欢迎下次使用" << endl;
                system("pause");
                return 0;
                break;
            default:
                break;
        }
    }
    system("pause");
    return 0;
}
```

效果图：

F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe

```
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
```

0
欢迎下次使用
请按任意键继续. . .

5、添加联系人

功能描述：

实现添加联系人功能，联系人上限为1000人，联系人信息包括（姓名、性别、年龄、联系电话、家庭住址）

添加联系人实现步骤：

- 设计联系人结构体
- 设计通讯录结构体
- main函数中创建通讯录
- 封装添加联系人函数
- 测试添加联系人功能

5.1 设计联系人结构体

联系人信息包括：姓名、性别、年龄、联系电话、家庭住址

设计如下：

```
#include <string> //string头文件
//联系人结构体
struct Person
{
    string m_Name; //姓名
    int m_Sex; //性别：1男 2女
    int m_Age; //年龄
    string m_Phone; //电话
    string m_Addr; //住址
};
```

5.2 设计通讯录结构体

设计时候可以在通讯录结构体中，维护一个容量为1000的存放联系人的数组，并记录当前通讯录中联系人数量
设计如下

```
#define MAX 1000 //最大人数

//通讯录结构体
struct Addressbooks
{
    struct Person personArray[MAX]; //通讯录中保存的联系人数组
    int m_Size; //通讯录中人员个数
};
```

5.3 main函数中创建通讯录

添加联系人函数封装好后，在main函数中创建一个通讯录变量，这个就是我们需要一直维护的通讯录

main函数起始位置添加：

```
//创建通讯录
Addressbooks abs;
//初始化通讯录中人数
abs.m_Size = 0;
```

5.4 封装添加联系人函数

思路：添加联系人前先判断通讯录是否已满，如果满了就不再添加，未满情况将新联系人信息逐个加入到通讯录

添加联系人代码：

```
//1、添加联系人信息
void addPerson(Addressbooks *abs)
{
    //判断电话本是否满了
    if (abs->m_Size == MAX)
    {
        cout << "通讯录已满，无法添加" << endl;
        return;
    }
    else
    {
        //姓名
        string name;
        cout << "请输入姓名：" << endl;
        cin >> name;
        abs->personArray[abs->m_Size].m_Name = name;

        cout << "请输入性别：" << endl;
        cout << "1 -- 男" << endl;
        cout << "2 -- 女" << endl;

        //性别
        int sex = 0;
        while (true)
        {
            cin >> sex;
            if (sex == 1 || sex == 2)
            {
                abs->personArray[abs->m_Size].m_Sex = sex;
                break;
            }
            cout << "输入有误，请重新输入";
        }

        //年龄
        cout << "请输入年龄：" << endl;
        int age = 0;
        cin >> age;
        abs->personArray[abs->m_Size].m_Age = age;

        //联系电话
        cout << "请输入联系电话：" << endl;
        string phone = "";
        cin >> phone;
        abs->personArray[abs->m_Size].m_Phone = phone;

        //家庭住址
        cout << "请输入家庭住址：" << endl;
        string address;
        cin >> address;
        abs->personArray[abs->m_Size].m_Addr = address;

        //更新通讯录人数
        abs->m_Size++;
    }
}
```

```
        cout << "添加成功" << endl;
        system("pause"); //请按任意键继续
        system("cls");   //清屏操作
    }
}
```

5.5 测试添加联系人功能

选择界面中，如果玩家选择了1，代表添加联系人，我们可以测试下该功能

在switch case 语句中，case1里添加：

```
case 1: //添加联系人
    addPerson(&abs);
    break;
```

测试效果如图：



6、显示联系人

功能描述：显示通讯录中已有的联系人信息

显示联系人实现步骤：

- 封装显示联系人函数
- 测试显示联系人功能

6.1 封装显示联系人函数

思路：判断如果当前通讯录中没有人员，就提示记录为空，人数大于0，显示通讯录中信息

显示联系人代码：

```
//2、显示所有联系人信息
void showPerson(Addressbooks * abs)
{
    if (abs->m_Size == 0)
    {
        cout << "当前记录为空" << endl;
    }
    else
    {
        for (int i = 0; i < abs->m_Size; i++)
        {
            cout << "姓名：" << abs->personArray[i].m_Name << "\t";
            cout << "性别：" << (abs->personArray[i].m_Sex == 1 ? "男" : "女") << "\t";
            cout << "年龄：" << abs->personArray[i].m_Age << "\t";
            cout << "电话：" << abs->personArray[i].m_Phone << "\t";
            cout << "住址：" << abs->personArray[i].m_Addr << endl;
        }
    }

    system("pause");
    system("cls");
}

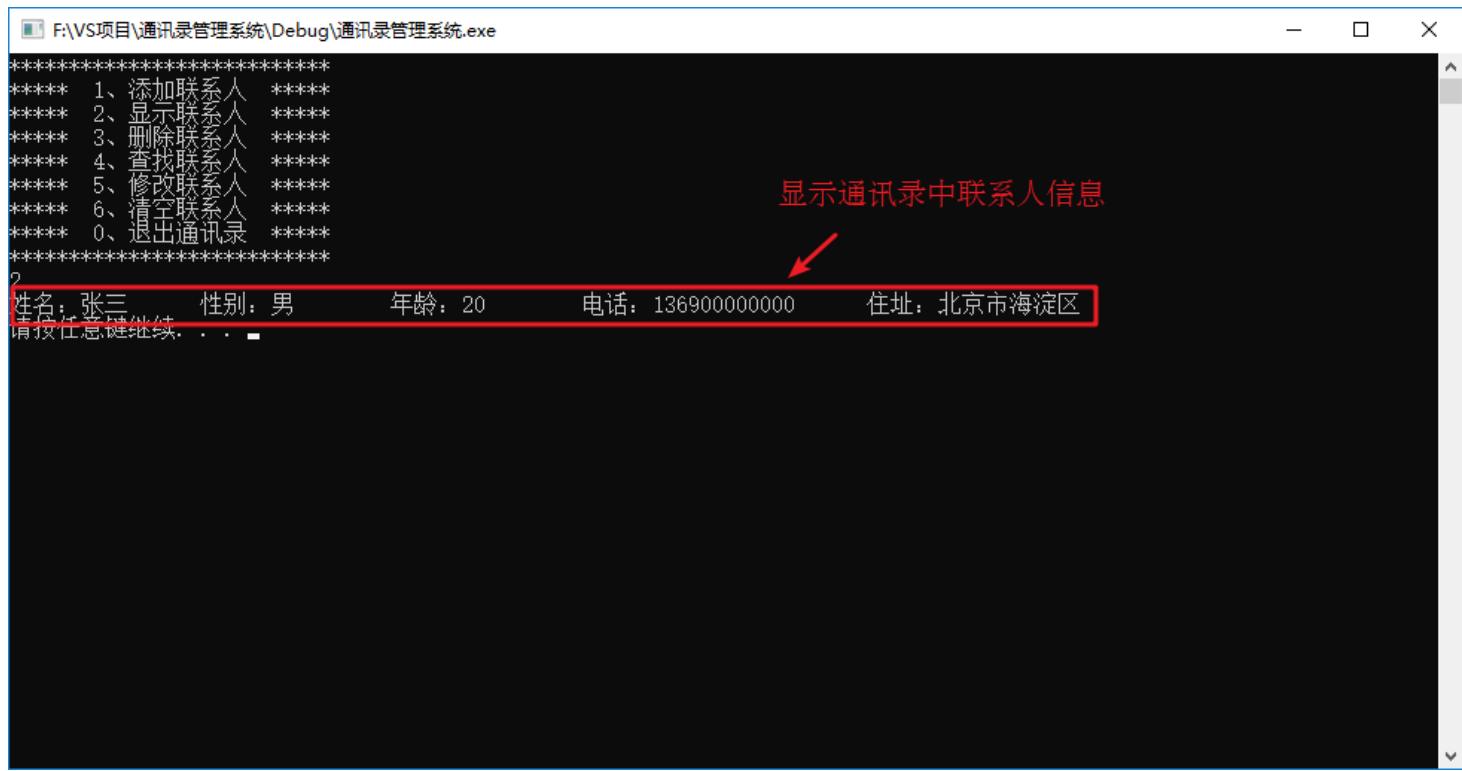
}
```

6.2 测试显示联系人功能

在switch case语句中，case 2 里添加

```
case 2: //显示联系人
    showPerson(&abs);
    break;
```

测试效果如图：



7、删除联系人

功能描述：按照姓名进行删除指定联系人

删除联系人实现步骤：

- 封装检测联系人是否存在
- 封装删除联系人函数
- 测试删除联系人功能

7.1 封装检测联系人是否存在

设计思路：

删除联系人前，我们需要先判断用户输入的联系人是否存在，如果存在删除，不存在提示用户没有要删除的联系人

因此我们可以把检测联系人是否存在封装成一个函数中，如果存在，返回联系人在通讯录中的位置，不存在返回-1

检测联系人是否存在代码：

```

//判断是否存在查询的人员，存在返回在数组中索引位置，不存在返回-1
int isExist(Addressbooks * abs, string name)
{
    for (int i = 0; i < abs->m_Size; i++)
    {
        if (abs->personArray[i].m_Name == name)
        {
            return i;
        }
    }
    return -1;
}

```

7.2 封装删除联系人函数

根据用户输入的联系人判断该通讯录中是否有此人

查找到进行删除，并提示删除成功

查不到提示查无此人。

```

//3、删除指定联系人信息
void deletePerson(Addressbooks * abs)
{
    cout << "请输入您要删除的联系人" << endl;
    string name;
    cin >> name;

    int ret = isExist(abs, name);
    if (ret != -1)
    {
        for (int i = ret; i < abs->m_Size; i++)
        {
            abs->personArray[i] = abs->personArray[i + 1];
        }
        abs->m_Size--;
        cout << "删除成功" << endl;
    }
    else
    {
        cout << "查无此人" << endl;
    }

    system("pause");
    system("cls");
}

```

上述代码中最后一个联系人不是重复了吗

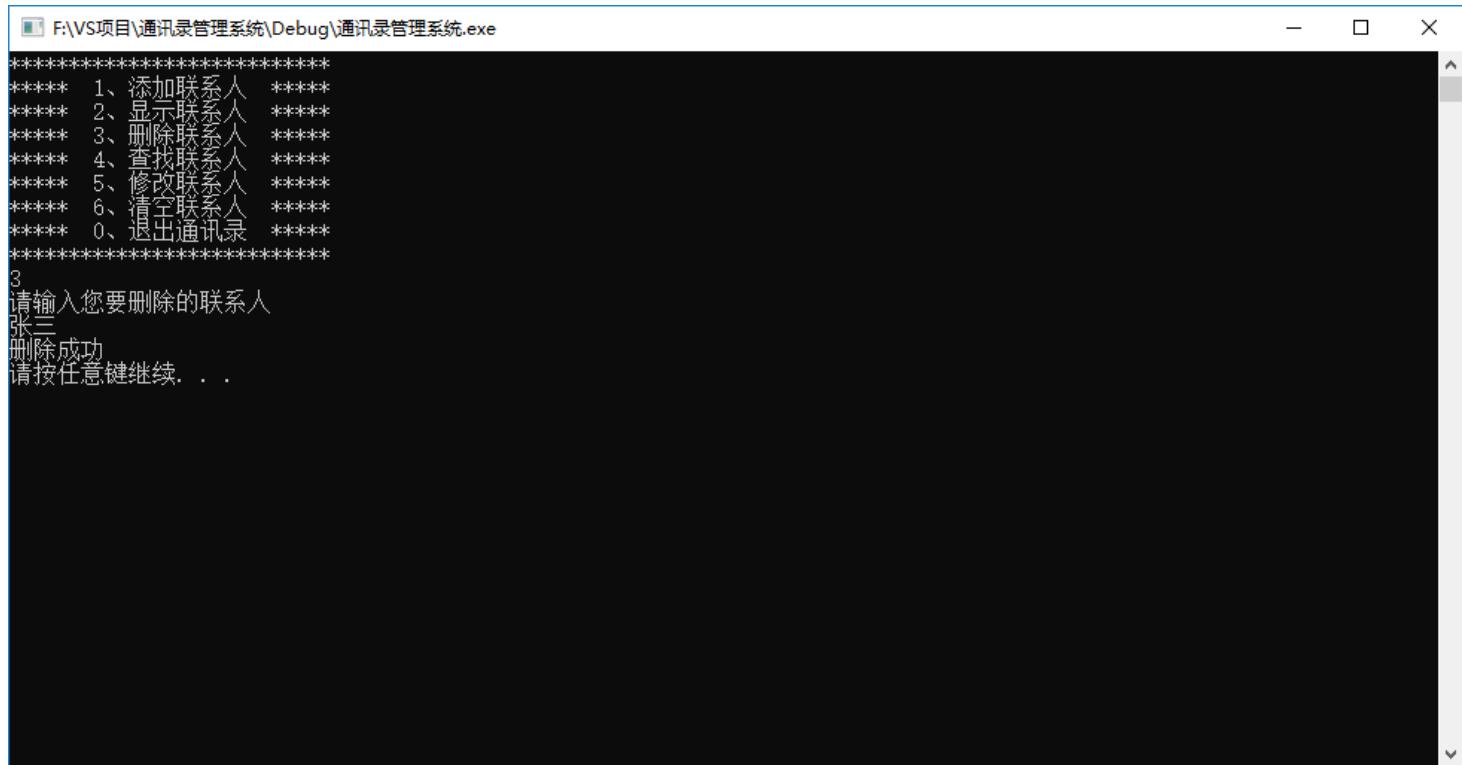
7.3 测试删除联系人功能

在switch case 语句中， case3里添加：

```
case 3: //删除联系人
    deletePerson(&abs);
    break;
```

测试效果如图：

存在情况：



不存在情况：

```
F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe
*****
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
*****
3
请输入您要删除的联系人
李四
查无此人
请按任意键继续. . .
```

8、查找联系人

功能描述：按照姓名查看指定联系人信息

查找联系人实现步骤

- 封装查找联系人函数
- 测试查找指定联系人

8.1 封装查找联系人函数

实现思路：判断用户指定的联系人是否存在，如果存在显示信息，不存在则提示查无此人。

查找联系人代码：

```
//4、查找指定联系人信息
void findPerson(Addressbooks * abs)
{
    cout << "请输入您要查找的联系人" << endl;
    string name;
    cin >> name;

    int ret = isExist(abs, name);
    if (ret != -1)
    {
        cout << "姓名: " << abs->personArray[ret].m_Name << "\t";
        cout << "性别: " << abs->personArray[ret].m_Sex << "\t";
        cout << "年龄: " << abs->personArray[ret].m_Age << "\t";
        cout << "电话: " << abs->personArray[ret].m_Phone << "\t";
        cout << "住址: " << abs->personArray[ret].m_Addr << endl;
    }
    else
    {
        cout << "查无此人" << endl;
    }

    system("pause");
    system("cls");
}
```

8.2 测试查找指定联系人

在switch case 语句中，case4里添加：

```
case 4: //查找联系人
    findPerson(&abs);
    break;
```

测试效果如图

存在情况：

```
F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe
*****
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
*****
4
请输入您要查找的联系人
张三
姓名: 张三 性别: 1 年龄: 20      电话: 13690000000      住址: 北京市海淀区
请按任意键继续. . .
```

不存在情况：

```
F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe
*****
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
*****
4
请输入您要查找的联系人
张小三
查无此人
请按任意键继续. . .
```

9、修改联系人

功能描述：按照姓名重新修改指定联系人

修改联系人实现步骤

- 封装修改联系人函数
- 测试修改联系人功能

9.1 封装修改联系人函数

实现思路：查找用户输入的联系人，如果查找成功进行修改操作，查找失败提示查无此人

修改联系人代码：

```
//5、修改指定联系人信息
void modifyPerson(Addressbooks * abs)
{
    cout << "请输入您要修改的联系人" << endl;
    string name;
    cin >> name;

    int ret = isExist(abs, name);
    if (ret != -1)
    {
        //姓名
        string name;
        cout << "请输入姓名：" << endl;
        cin >> name;
        abs->personArray[ret].m_Name = name;

        cout << "请输入性别：" << endl;
        cout << "1 -- 男" << endl;
        cout << "2 -- 女" << endl;

        //性别
        int sex = 0;
        while (true)
        {
            cin >> sex;
            if (sex == 1 || sex == 2)
            {
                abs->personArray[ret].m_Sex = sex;
                break;
            }
            cout << "输入有误，请重新输入";
        }

        //年龄
        cout << "请输入年龄：" << endl;
        int age = 0;
        cin >> age;
        abs->personArray[ret].m_Age = age;

        //联系电话
        cout << "请输入联系电话：" << endl;
        string phone = "";
        cin >> phone;
        abs->personArray[ret].m_Phone = phone;

        //家庭住址
        cout << "请输入家庭住址：" << endl;
        string address;
        cin >> address;
        abs->personArray[ret].m_Addr = address;

        cout << "修改成功" << endl;
    }
    else
    {
```

```
        cout << "查无此人" << endl;
    }

    system("pause");
    system("cls");

}
```

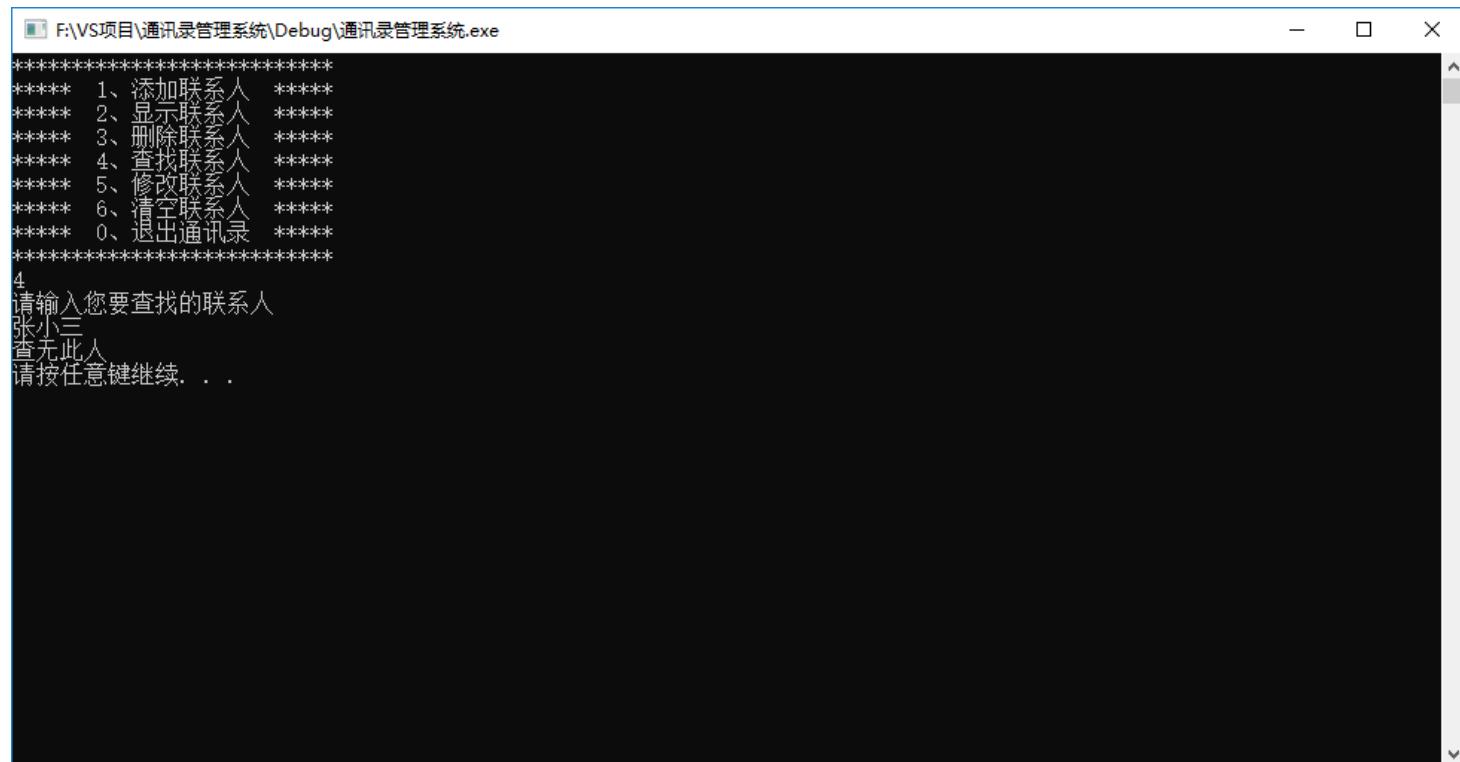
9.2 测试修改联系人功能

在switch case 语句中，case 5里添加：

```
case 5: //修改联系人
    modifyPerson(&abs);
    break;
```

测试效果如图：

查不到指定联系人情况：



查找到联系人，并修改成功：

```
F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe
*****
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
*****
5
请输入您要修改的联系人
张三
请输入姓名：
张三丰
请输入性别：
1 -- 男
2 -- 女
1
请输入年龄：
25
请输入联系电话：
13690000000
请输入家庭住址：
北京市朝阳区
修改成功
请按任意键继续. . .
```

再次查看通讯录，确认修改完毕

```
F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe
*****
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
*****
2
姓名：张三丰 性别：男 年龄：25 电话：13690000000 住址：北京市朝阳区
请按任意键继续. . .
```

10、清空联系人

功能描述：清空通讯录中所有信息

清空联系人实现步骤

- 封装清空联系人函数
- 测试清空联系人

10.1 封装清空联系人函数

实现思路：将通讯录所有联系人信息清除掉，只要将通讯录记录的联系人数量置为0，做逻辑清空即可。

清空联系人代码：

```
//6、清空所有联系人
void cleanPerson(Addressbooks * abs)
{
    abs->m_Size = 0;
    cout << "通讯录已清空" << endl;
    system("pause");
    system("cls");
}
```

10.2 测试清空联系人

在switch case 语句中，case 6 里添加：

```
case 6: //清空联系人
    cleanPerson(&abs);
    break;
```

测试效果如图：

清空通讯录

```
F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe
*****
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
*****
6
通讯录已清空
请按任意键继续. . .
```

再次查看信息，显示记录为空

```
F:\VS项目\通讯录管理系统\Debug\通讯录管理系统.exe
*****
***** 1、添加联系人 *****
***** 2、显示联系人 *****
***** 3、删除联系人 *****
***** 4、查找联系人 *****
***** 5、修改联系人 *****
***** 6、清空联系人 *****
***** 0、退出通讯录 *****
*****
2
当前记录为空
请按任意键继续. . .
```

至此，通讯录管理系统完成！

职工管理系统

1、管理系统需求

职工管理系统可以用来管理公司内所有员工的信息

本教程主要利用C++来实现一个基于多态的职工管理系统

公司中职工分为三类：普通员工、经理、老板，显示信息时，需要显示职工编号、职工姓名、职工岗位、以及职责

普通员工职责：完成经理交给的任务

经理职责：完成老板交给的任务，并下发任务给员工

老板职责：管理公司所有事务

管理系统中需要实现的功能如下：

- 退出管理程序：退出当前管理系统
- 增加职工信息：实现批量添加职工功能，将信息录入到文件中，职工信息为：职工编号、姓名、部门编号
- 显示职工信息：显示公司内部所有职工的信息
- 删除离职职工：按照编号删除指定的职工
- 修改职工信息：按照编号修改职工个人信息
- 查找职工信息：按照职工的编号或者职工的姓名进行查找相关的人员信息
- 按照编号排序：按照职工编号，进行排序，排序规则由用户指定
- 清空所有文档：清空文件中记录的所有职工信息（清空前需要再次确认，防止误删）

系统界面效果图如下：

```
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****
请输入您的选择:
```

需根据用户不同的选择，完成不同的功能！

2、创建项目

创建项目步骤如下：

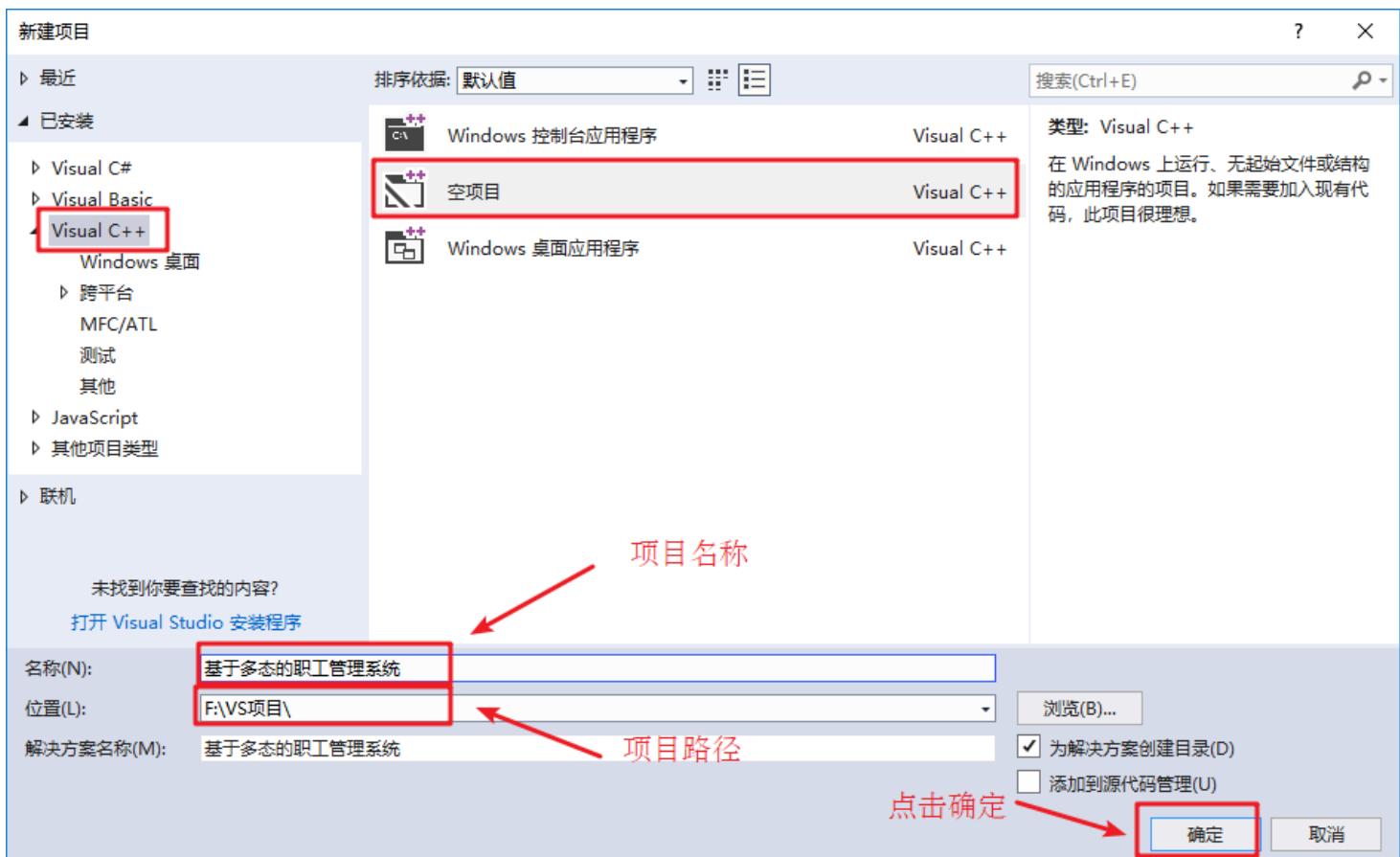
- 创建新项目
- 添加文件

2.1 创建项目

打开vs2017后，点击创建新项目，创建新的C++项目

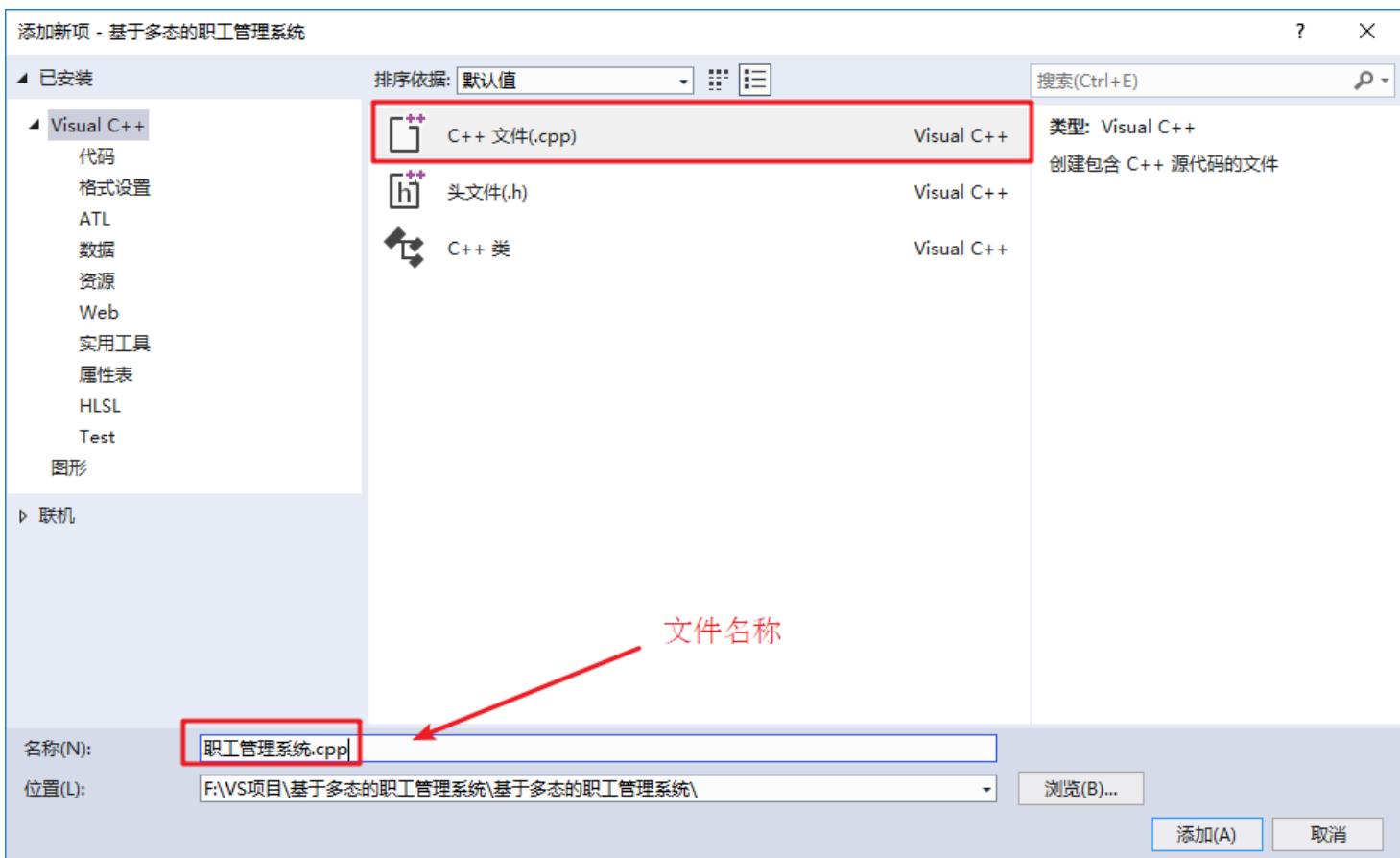
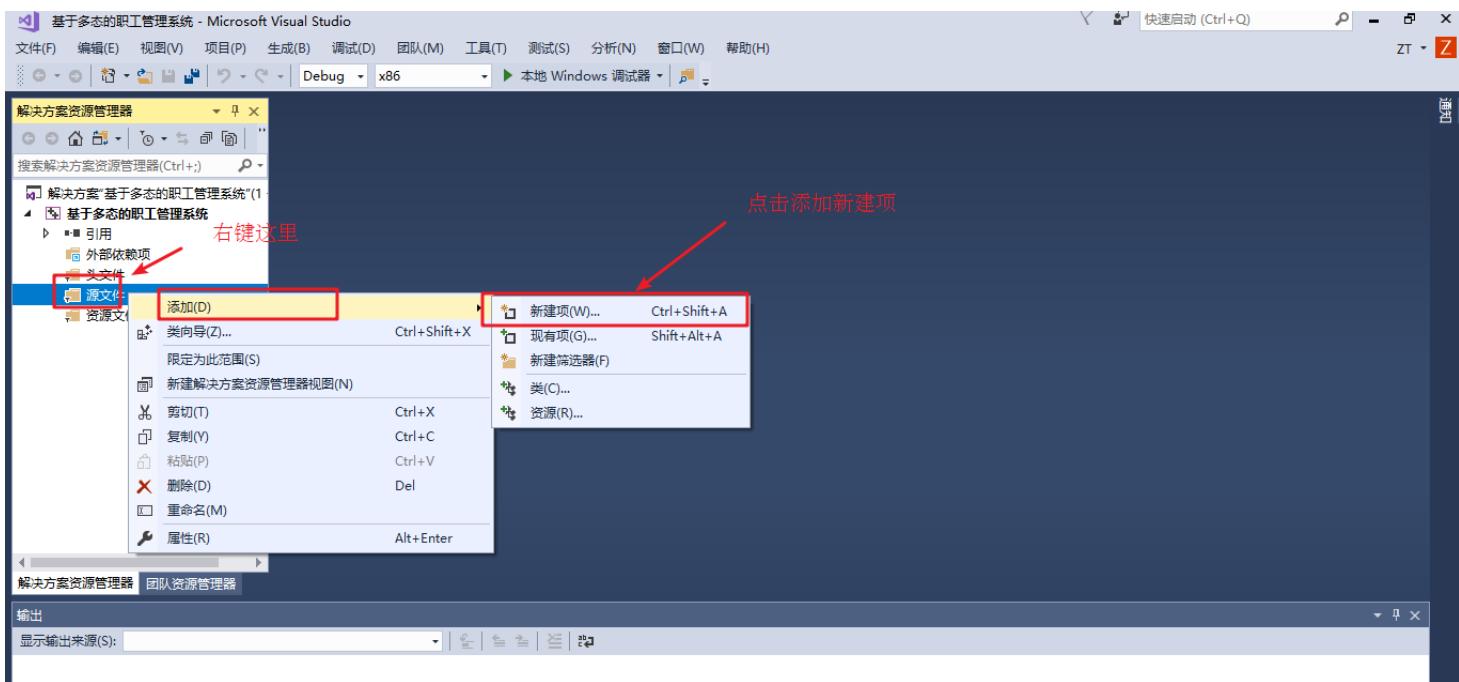


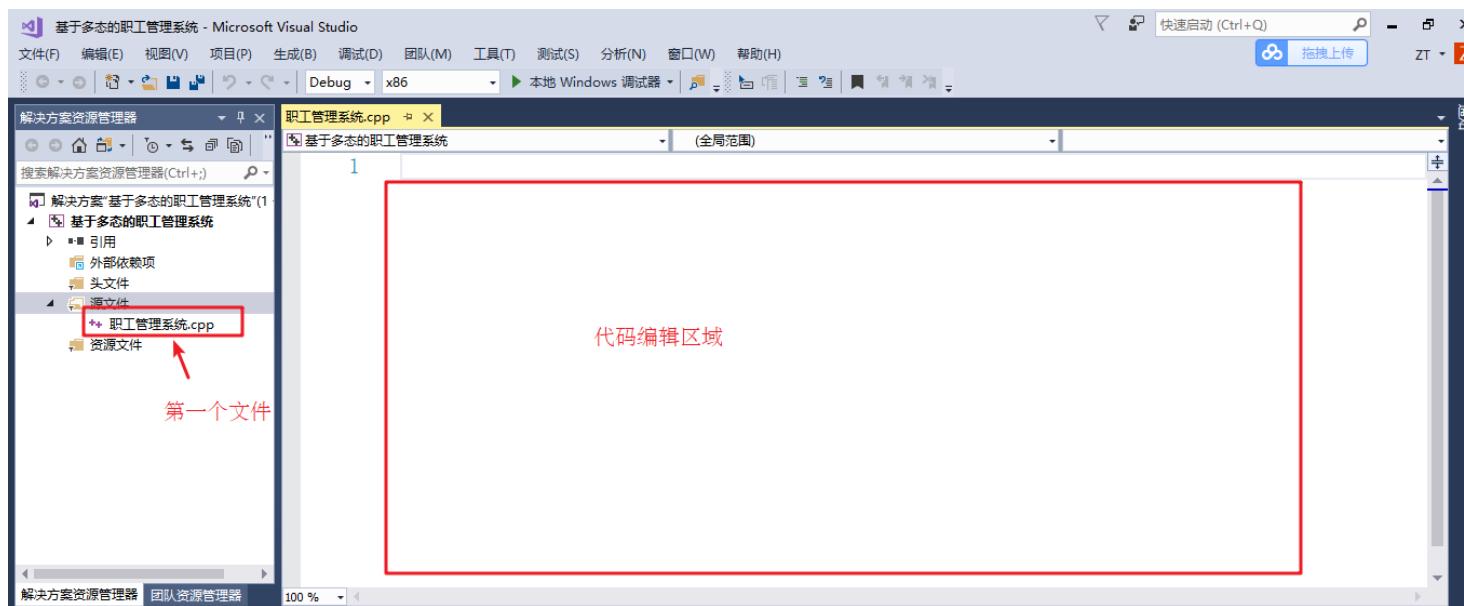
填写项目名称以及项目路径，点击确定



2.2 添加文件

右键源文件，进行添加文件操作





至此，项目已创建完毕

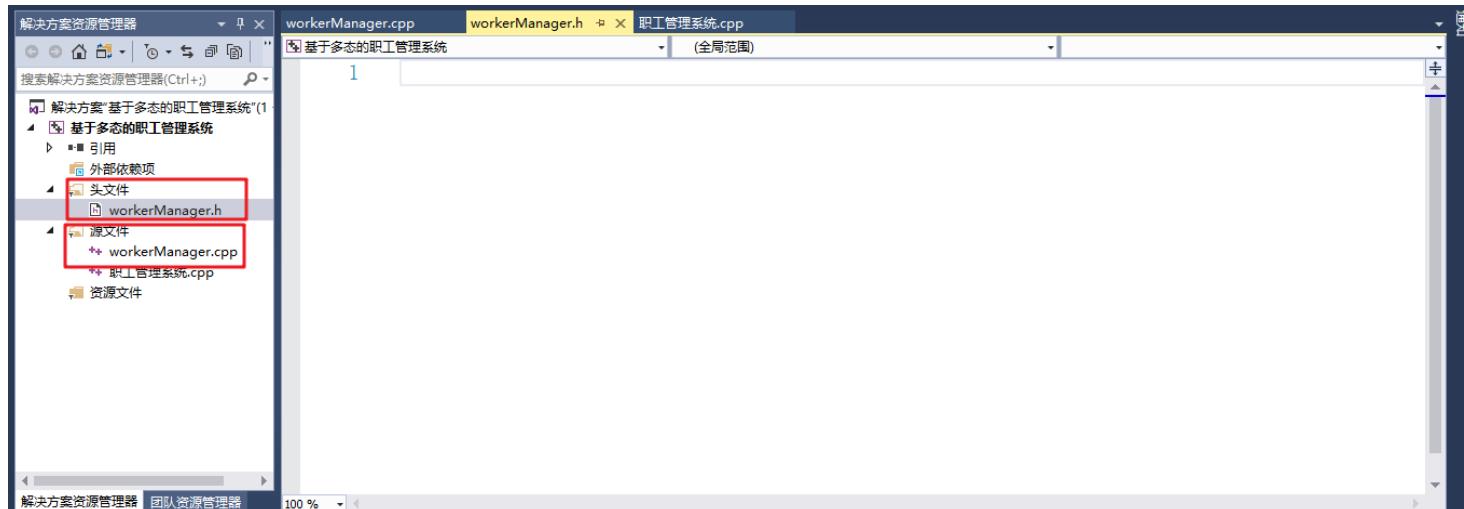
3、创建管理类

管理类负责的内容如下：

- 与用户的沟通菜单界面
- 对职工增删改查的操作
- 与文件的读写交互

3.1 创建文件

在头文件和源文件的文件夹下分别创建workerManager.h 和 workerManager.cpp文件



3.2 头文件实现

在workerManager.h中设计管理类

代码如下：

```
#pragma once
#include<iostream>
using namespace std;

class WorkerManager
{
public:

    //构造函数
    WorkerManager();

    //析构函数
    ~WorkerManager();

};

};
```

3.3 源文件实现

在workerManager.cpp中将构造和析构函数空实现补全

```
#include "workerManager.h"

WorkerManager::WorkerManager()
{
}

WorkerManager::~WorkerManager()
{
}
```

至此职工管理类以创建完毕

4、菜单功能

功能描述：与用户的沟通界面

4.1 添加成员函数

在管理类workerManager.h中添加成员函数 void Show_Menu();

```
6 class WorkerManager
7 {
8 public:
9
10 //构造函数
11 WorkerManager();
12
13 //展示菜单
14 void Show_Menu();
15
16 //析构函数
17 ~WorkerManager();
18
19 };
```

4.2 菜单功能实现

在管理类workerManager.cpp中实现 Show_Menu() 函数

```
void WorkerManager::Show_Menu()
{
    cout << "***** 欢迎使用职工管理系统! *****" << endl;
    cout << "***** 0.退出管理程序" << endl;
    cout << "***** 1.增加职工信息" << endl;
    cout << "***** 2.显示职工信息" << endl;
    cout << "***** 3.删除离职职工" << endl;
    cout << "***** 4.修改职工信息" << endl;
    cout << "***** 5.查找职工信息" << endl;
    cout << "***** 6.按照编号排序" << endl;
    cout << "***** 7.清空所有文档" << endl;
    cout << "*****" << endl;
    cout << endl;
```

4.3 测试菜单功能

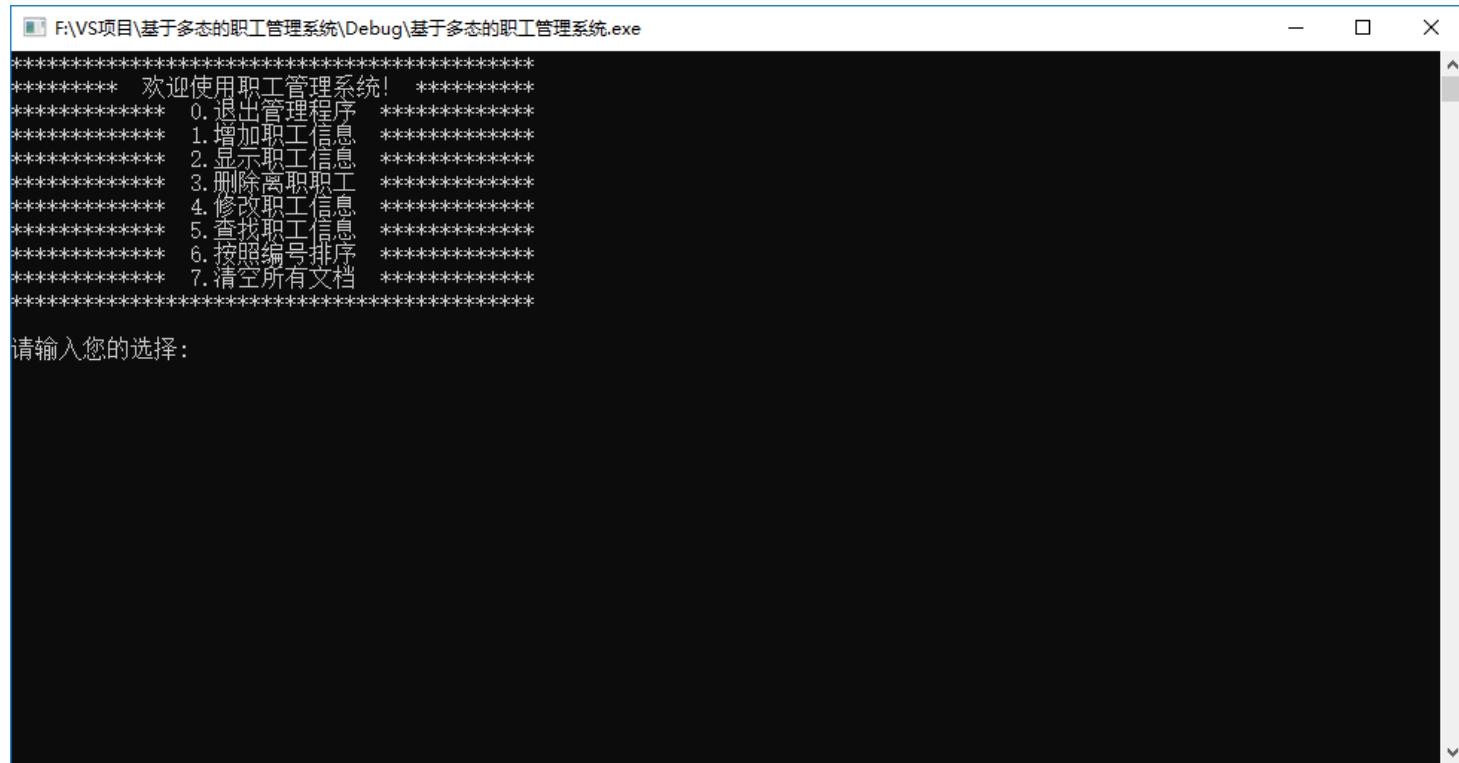
在职工管理系统.cpp中测试菜单功能

代码：

```
#include<iostream>
using namespace std;
#include "workerManager.h"

int main() {
    WorkerManager wm;
    wm.Show_Menu();
    system("pause");
    return 0;
}
```

运行效果如图：



5、退出功能

5.1 提供功能接口

在main函数中提供分支选择，提供每个功能接口

代码：

```

int main() {

    WorkerManager wm;
    int choice = 0;
    while (true)
    {
        //展示菜单
        wm.Show_Menu();
        cout << "请输入您的选择：" << endl;
        cin >> choice;

        switch (choice)
        {
            case 0: //退出系统
                break;
            case 1: //添加职工
                break;
            case 2: //显示职工
                break;
            case 3: //删除职工
                break;
            case 4: //修改职工
                break;
            case 5: //查找职工
                break;
            case 6: //排序职工
                break;
            case 7: //清空文件
                break;
            default:
                system("cls");
                break;
        }
    }

    system("pause");
    return 0;
}

```

5.2 实现退出功能

在workerManager.h中提供退出系统的成员函数 void exitSystem();

在workerManager.cpp中提供具体的功能实现

```

void WorkerManager::exitSystem()
{
    cout << "欢迎下次使用" << endl;
    system("pause");
    exit(0);
}

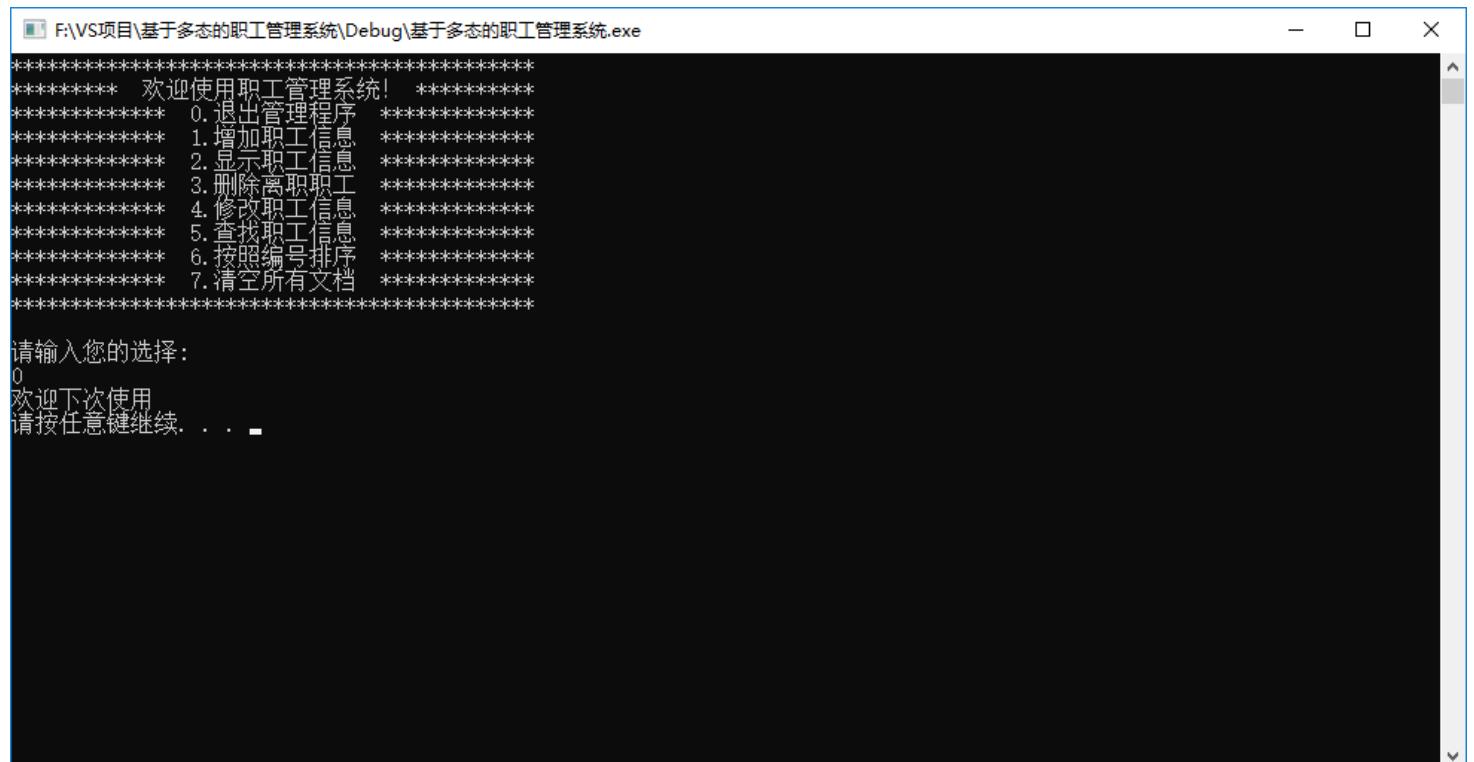
```

5.3 测试功能

在main函数分支0选项中，调用退出程序的接口

```
switch (choice)
{
    case 0: //退出系统
        wm.exitSystem();
        break;
    case 1: //添加职工
        break;
    case 2: //显示职工
        break;
    case 3: //删除职工
        break;
    case 4: //修改职工
        break;
    case 5: //查找职工
        break;
    case 6: //排序职工
        break;
    case 7: //清空文件
        break;
    default:
        cout << " -1 -" << endl;
}
```

运行测试效果如图：



6、创建职工类

6.1 创建职工抽象类

职工的分类为：普通员工、经理、老板

将三种职工抽象到一个类（worker）中，利用多态管理不同职工种类

职工的属性为：职工编号、职工姓名、职工所在部门编号

职工的行为为：岗位职责信息描述，获取岗位名称

头文件文件夹下 创建文件worker.h 文件并且添加如下代码：

```
#pragma once
#include<iostream>
#include<string>
using namespace std;

//职工抽象基类
class Worker
{
public:

    //显示个人信息
    virtual void showInfo() = 0;
    //获取岗位名称
    virtual string getDeptName() = 0;

    int m_Id; //职工编号
    string m_Name; //职工姓名
    int m_DeptId; //职工所在部门名称编号
};
```

6.2 创建普通员工类

普通员工类继承职工抽象类，并重写父类中纯虚函数

在头文件和源文件的文件夹下分别创建employee.h 和 employee.cpp文件

employee.h中代码如下：

```

#pragma once
#include<iostream>
using namespace std;
#include "worker.h"

//员工类
class Employee :public Worker
{
public:

    //构造函数
    Employee(int id, string name, int dId);

    //显示个人信息
    virtual void showInfo();

    //获取职工岗位名称
    virtual string getDeptName();
};


```

employee.cpp中代码如下：

```

#include "employee.h"

Employee::Employee(int id, string name, int dId)
{
    this->m_Id = id;
    this->m_Name = name;
    this->m_DeptId = dId;
}

void Employee::showInfo()
{
    cout << "职工编号: " << this->m_Id
        << " \t职工姓名: " << this->m_Name
        << " \t岗位: " << this->getDeptName()
        << " \t岗位职责: 完成经理交给的任务" << endl;
}

string Employee::getDeptName()
{
    return string("员工");
}

```

6.3 创建经理类

经理类继承职工抽象类，并重写父类中纯虚函数，和普通员工类似

在头文件和源文件的文件夹下分别创建manager.h 和 manager.cpp文件

manager.h中代码如下：

```
#pragma once
#include<iostream>
using namespace std;
#include "worker.h"

//经理类
class Manager :public Worker
{
public:

    Manager(int id, string name, int dId);

    //显示个人信息
    virtual void showInfo();

    //获取职工岗位名称
    virtual string getDeptName();
};
```

manager.cpp中代码如下：

```
#include "manager.h"

Manager::Manager(int id, string name, int dId)
{
    this->m_Id = id;
    this->m_Name = name;
    this->m_DeptId = dId;

}

void Manager::showInfo()
{
    cout << "职工编号：" << this->m_Id
        << "\t职工姓名：" << this->m_Name
        << "\t岗位：" << this->getDeptName()
        << "\t岗位职责：完成老板交给的任务，并下发任务给员工" << endl;
}

string Manager::getDeptName()
{
    return string("经理");
}
```

6.4 创建老板类

老板类**继承**职工抽象类，并重写父类中纯虚函数，和普通员工类似

在头文件和源文件的文件夹下分别创建boss.h 和 boss.cpp文件

boss.h中代码如下：

```
#pragma once
#include<iostream>
using namespace std;
#include "worker.h"

//老板类
class Boss :public Worker
{
public:
    Boss(int id, string name, int dId);

    //显示个人信息
    virtual void showInfo();

    //获取职工岗位名称
    virtual string getDeptName();
};
```

boss.cpp中代码如下：

```
#include "boss.h"

Boss::Boss(int id, string name, int dId)
{
    this->m_Id = id;
    this->m_Name = name;
    this->m_DeptId = dId;
}

void Boss::showInfo()
{
    cout << "职工编号: " << this->m_Id
        << " \t职工姓名: " << this->m_Name
        << " \t岗位: " << this->getDeptName()
        << " \t岗位职责: 管理公司所有事务" << endl;
}

string Boss::getDeptName()
{
    return string("总裁");
}
```

6.5 测试多态

在职工管理系统.cpp中添加测试函数，并且运行能够产生多态

测试代码如下：

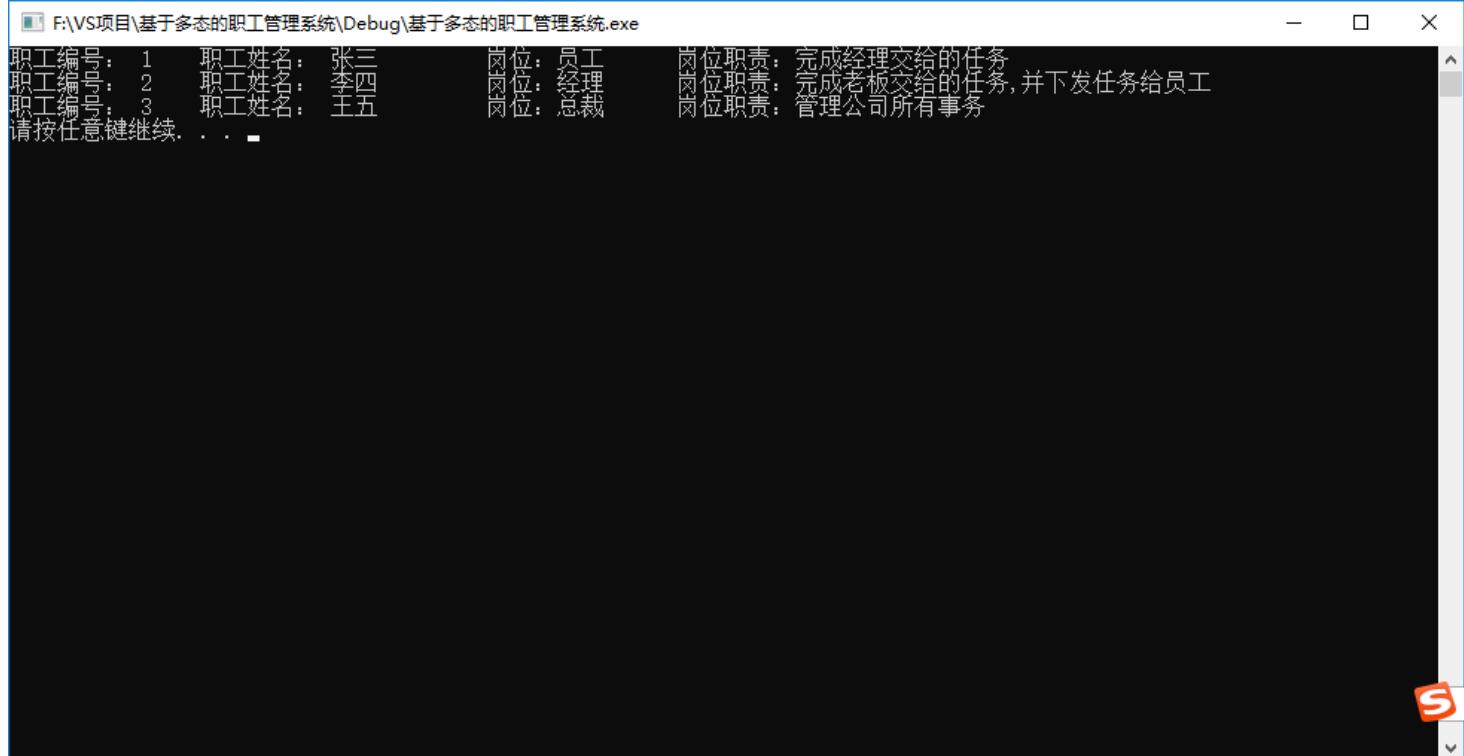
```
#include "worker.h"
#include "employee.h"
#include "manager.h"
#include "boss.h"

void test()
{
    Worker * worker = NULL;
    worker = new Employee(1, "张三", 1);
    worker->showInfo();
    delete worker;

    worker = new Manager(2, "李四", 2);
    worker->showInfo();
    delete worker;

    worker = new Boss(3, "王五", 3);
    worker->showInfo();
    delete worker;
}
```

运行效果如图：



测试成功后，测试代码可以注释保留，或者选择删除

7、添加职工

功能描述：批量添加职工，并且保存到文件中

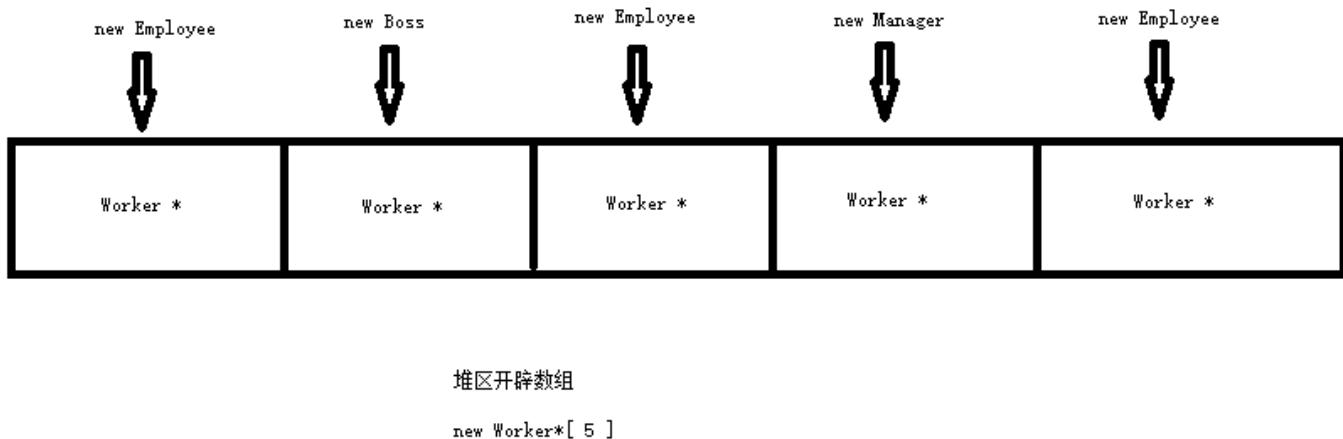
7.1 功能分析

分析：

用户在批量创建时，可能会创建不同种类的职工

如果想将所有不同种类的员工都放入到一个数组中，可以将所有员工的指针维护到一个数组里

如果想在程序中维护这个不定长度的数组，可以将数组创建到堆区，并利用Worker **的指针维护



7.2 功能实现

在WokerManager.h头文件中添加成员属性 代码：

```
//记录文件中的人数个数  
int m_EmpNum;  
  
//员工数组的指针  
Worker ** m_EmpArray;
```

在WorkerManager构造函数中初始化属性

```
WorkerManager::WorkerManager()
{
    //初始化人数
    this->m_EmpNum = 0;

    //初始化数组指针
    this->m_EmpArray = NULL;
}
```

在workerManager.h中添加成员函数

```
//增加职工
void Add_Emp();
```

workerManager.cpp中实现该函数

```
//增加职工
void WorkerManager::Add_Emp()
{
    cout << "请输入增加职工数量: " << endl;

    int addNum = 0;
    cin >> addNum;

    if (addNum > 0)
    {
        //计算新空间大小
        int newSize = this->m_EmpNum + addNum;

        //开辟新空间
        Worker ** newSpace = new Worker*[newSize];

        //将原空间下内容存放到新空间下
        if (this->m_EmpArray != NULL)
        {
            for (int i = 0; i < this->m_EmpNum; i++)
            {
                newSpace[i] = this->m_EmpArray[i];
            }
        }

        //输入新数据
        for (int i = 0; i < addNum; i++)
        {
            int id;
            string name;
            int dSelect;

            cout << "请输入第 " << i + 1 << " 个新职工编号: " << endl;
            cin >> id;

            cout << "请输入第 " << i + 1 << " 个新职工姓名: " << endl;
            cin >> name;

            cout << "请选择该职工的岗位: " << endl;
            cout << "1、普通职工" << endl;
            cout << "2、经理" << endl;
            cout << "3、老板" << endl;
            cin >> dSelect;

            Worker * worker = NULL;
            switch (dSelect)
            {
                case 1: //普通员工
                    worker = new Employee(id, name, 1);
                    break;
                case 2: //经理
                    worker = new Manager(id, name, 2);
                    break;
            }

            newSpace[i] = worker;
        }

        delete[] this->m_EmpArray;
        this->m_EmpArray = newSpace;
        this->m_EmpNum += addNum;
    }
}
```

```

        break;
    case 3: //老板
        worker = new Boss(id, name, 3);
        break;
    default:
        break;
    }

    newSpace[this->m_EmpNum + i] = worker;
}

//释放原有空间
delete[] this->m_EmpArray;

//更改新空间的指向
this->m_EmpArray = newSpace;

//更新新的个数
this->m_EmpNum = newSize;

//提示信息
cout << "成功添加" << addNum << "名新职工！" << endl;
}
else
{
    cout << "输入有误" << endl;
}

system("pause");
system("cls");
}

```

在WorkerManager.cpp的析构函数中，释放堆区数据

```

WorkerManager::~WorkerManager()
{
    if (this->m_EmpArray != NULL)
    {
        delete[] this->m_EmpArray;
    }
}

```

7.3 测试添加

在main函数分支 1 选项中，调用添加职工接口

```
switch (choice)
{
case 0: //退出系统
    wm.exitSystem();
    break;
case 1: //添加职工
    wm.Add_Emp();
    break;
case 2: //显示职工
    break;
case 3: //删除职工
    break;
case 4: //修改职工
    break;
case 5: //查找职工
    break;
case 6: //排序职工
    break;
case 7: //清空文件
    break;
default:
    system("cls");
    break;
}
```

效果如图：

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****
请输入您的选择:
1
请输入增加职工数量:
1
请输入第 1 个新职工编号:
1
请输入第 1 个新职工姓名:
张三
请选择该职工的岗位:
1、普通职工
2、经理
3、老板
1
成功添加1名新职工!
请按任意键继续. . .
```

至此，添加职工到程序中功能实现完毕

8、文件交互 - 写文件

功能描述：对文件进行读写

在上一个添加功能中，我们只是将所有的数据添加到了内存中，一旦程序结束就无法保存了

因此文件管理类中需要一个与文件进行交互的功能，对于文件进行读写操作

8.1 设定文件路径

首先我们将文件路径，在workerManager.h中添加宏常量，并且包含头文件 fstream

```
#include <fstream>
#define FILENAME "empFile.txt"
```

8.2 成员函数声明

在workerManager.h中类里添加成员函数 void save()

```
//保存文件
void save();
```

8.3 保存文件功能实现

```
void WorkerManager::save()
{
    ofstream ofs;
    ofs.open(FILENAME, ios::out);

    for (int i = 0; i < this->m_EmpNum; i++)
    {
        ofs << this->m_EmpArray[i]->m_Id << " "
            << this->m_EmpArray[i]->m_Name << " "
            << this->m_EmpArray[i]->m_DeptId << endl;
    }

    ofs.close();
}
```

8.4 保存文件功能测试

在添加职工功能中添加成功后添加保存文件函数

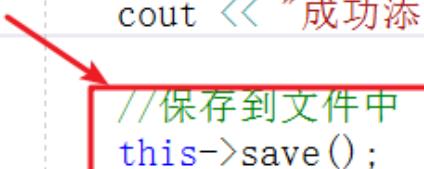
```
    newSpace[this->m_EmpNum + i] = worker;
}

//释放原有空间
delete[] this->m_EmpArray;

//更改新空间的指向
this->m_EmpArray = newSpace;

//更新新的个数
this->m_EmpNum = newSize;

//提示信息
cout << "成功添加" << addNum << "名新职工!" << endl;



```
//保存到文件中
this->save();
```

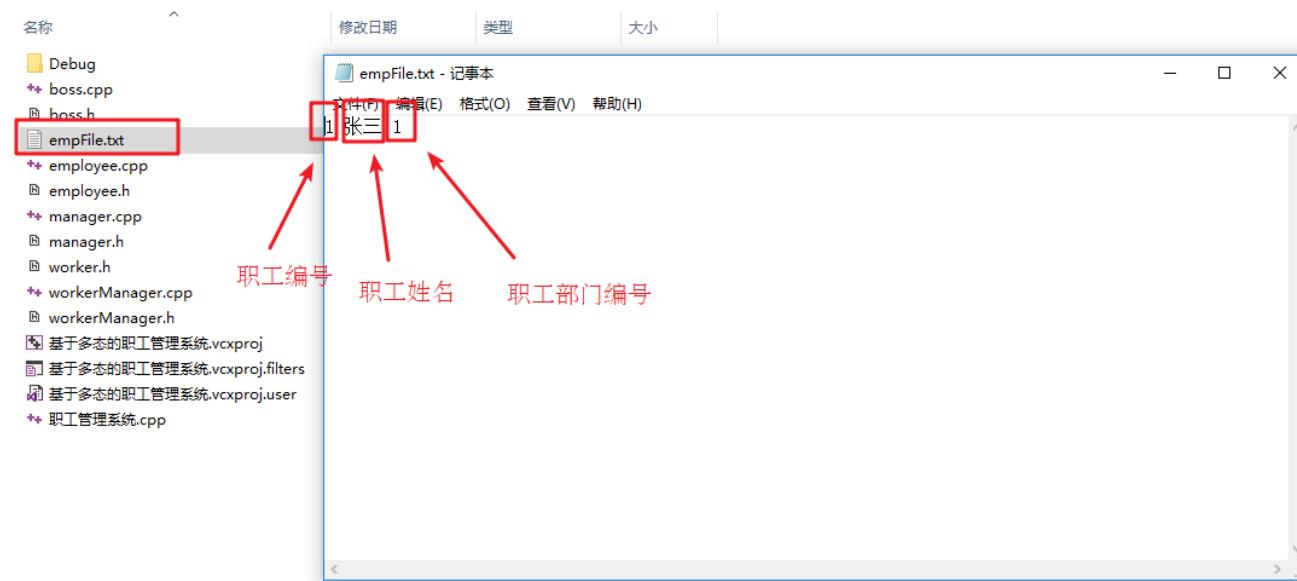

```

再次运行代码，添加职工

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
****

请输入您的选择:
1
请输入增加职工数量:
1
请输入第 1 个新职工编号:
1
请输入第 1 个新职工姓名:
张三
请选择该职工的岗位:
1、普通职工
2、经理
3、老板
1
成功添加1名新职工!
请按任意键继续. . .
```

同级目录下多出文件，并且保存了添加的信息



9、文件交互 - 读文件

功能描述：将文件中的内容读取到程序中

虽然我们实现了添加职工后保存到文件的操作，但是每次开始运行程序，并没有将文件中数据读取到程序中

而我们的程序功能中还有清空文件的需求

因此构造函数初始化数据的情况分为三种

1. 第一次使用，文件未创建
2. 文件存在，但是数据被用户清空
3. 文件存在，并且保存职工的所有数据

9.1 文件未创建

在workerManager.h中添加新的成员属性 m_FileIsEmpty 标志文件是否为空

```
//标志文件是否为空  
bool m_FileIsEmpty;
```

修改WorkerManager.cpp中构造函数代码

```
WorkerManager::WorkerManager()  
{  
    ifstream ifs;  
    ifs.open(FILENAME, ios::in);  
  
    //文件不存在情况  
    if (!ifs.is_open())  
    {  
        cout << "文件不存在" << endl; //测试输出  
        this->m_EmpNum = 0; //初始化人数  
        this->m_FileIsEmpty = true; //初始化文件为空标志  
        this->m_EmpArray = NULL; //初始化数组  
        ifs.close(); //关闭文件  
        return;  
    }  
}
```

删除文件后，测试文件不存在时初始化数据功能

9.2 文件存在且数据为空

在workerManager.cpp中的构造函数追加代码：

```
//文件存在，并且没有记录
char ch;
ifs >> ch;
if (ifs.eof())
{
    cout << "文件为空!" << endl;
    this->m_EmpNum = 0;
    this->m_FileIsEmpty = true;
    this->m_EmpArray = NULL;
    ifs.close();
    return;
}
```

追加代码位置如图：

```
//文件不存在情况
if (!ifs.is_open())
{
    cout << "文件不存在" << endl;
    this->m_EmpNum = 0; //初始化人数
    this->m_FileIsEmpty = true; //初始化文件为空标志
    this->m_EmpArray = NULL; //初始化数组
    ifs.close(); //关闭文件
    return;
}

//文件存在，并且没有记录
char ch;
ifs >> ch;
if (ifs.eof())
{
    cout << "文件为空!" << endl;
    this->m_EmpNum = 0;
    this->m_FileIsEmpty = true;
    this->m_EmpArray = NULL;
    ifs.close();
    return;
}
```

将文件创建后清空文件内容，并测试该情况下初始化功能

我们发现文件不存在或者为空清空 m_FileIsEmpty 判断文件是否为空的标志都为真，那何时为假？

成功添加职工后，应该更改文件不为空的标志

在 void WorkerManager::Add_Emp() 成员函数中添加：

```
//更新职工不为空标志
this->m_FileIsEmpty = false;

//更改新空间的指向
this->m_EmpArray = newSpace;

//更新新的个数
this->m_EmpNum = newSize;

//更新职工不为空标志
this->m_FileIsEmpty = false;

//提示信息
cout << "成功添加" << addNum << "名新职工!" << endl;

//保存到文件中
this->save();
```

9.3 文件存在且保存职工数据

9.3.1 获取记录的职工人数

在workerManager.h中添加成员函数 int get_EmpNum();

```
//统计人数
int get_EmpNum();
```

workerManager.cpp中实现

```
int WorkerManager::get_EmpNum()
{
    ifstream ifs;
    ifs.open(FILENAME, ios::in);

    int id;
    string name;
    int dId;

    int num = 0;

    while (ifs >> id && ifs >> name && ifs >> dId)
    {
        //记录人数
        num++;
    }
    ifs.close();

    return num;
}
```

在workerManager.cpp构造函数中继续追加代码：

```
int num = this->get_EmpNum();
cout << "职工个数为：" << num << endl; //测试代码
this->m_EmpNum = num; //更新成员属性
```

手动添加一些职工数据，测试获取职工数量函数



```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
职工个数为: 3 ← 测试数据输出
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****
请输入您的选择:
```

9.3.2 初始化数组

根据职工的数据以及职工数据，初始化workerManager中的Worker ** m_EmpArray 指针

在WorkerManager.h中添加成员函数 void init_Emp();

```
//初始化员工
void init_Emp();
```

在WorkerManager.cpp中实现

```

void WorkerManager::init_Emp()
{
    ifstream ifs;
    ifs.open(FILENAME, ios::in);

    int id;
    string name;
    int dId;

    int index = 0;
    while (ifs >> id && ifs >> name && ifs >> dId)
    {
        Worker * worker = NULL;
        //根据不同的部门Id创建不同对象
        if (dId == 1) // 1普通员工
        {
            worker = new Employee(id, name, dId);
        }
        else if (dId == 2) //2经理
        {
            worker = new Manager(id, name, dId);
        }
        else //总裁
        {
            worker = new Boss(id, name, dId);
        }
        //存放在数组中
        this->m_EmpArray[index] = worker;
        index++;
    }
}

```

在workerManager.cpp构造函数中追加代码

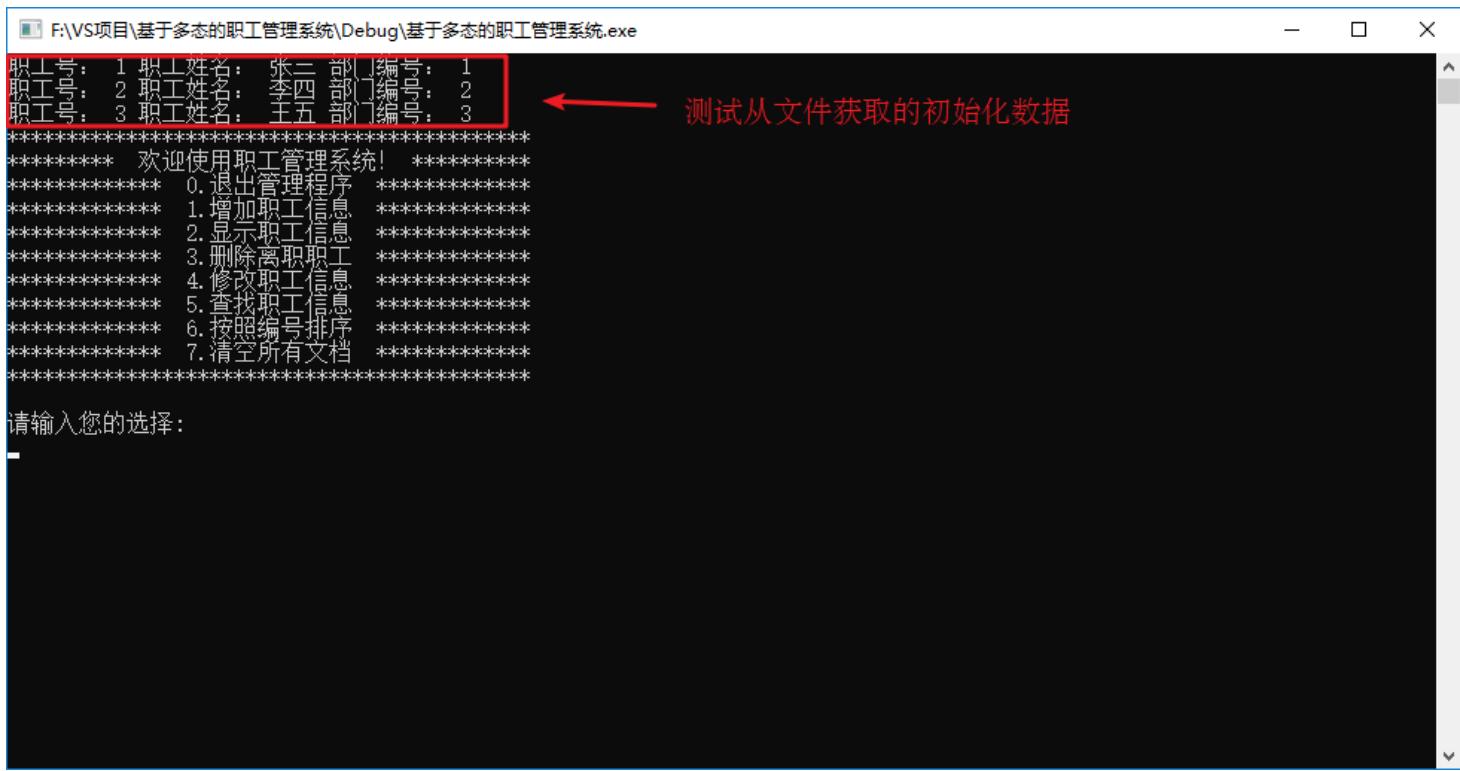
```

//根据职工数创建数组
this->m_EmpArray = new Worker *[this->m_EmpNum];
//初始化职工
init_Emp();

//测试代码
for (int i = 0; i < m_EmpNum; i++)
{
    cout << "职工号: " << this->m_EmpArray[i]->m_Id
        << " 职工姓名: " << this->m_EmpArray[i]->m_Name
        << " 部门编号: " << this->m_EmpArray[i]->m_DeptId << endl;
}

```

运行程序，测试从文件中获取的数据



```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
职工号: 1 职工姓名: 张三 部门编号: 1
职工号: 2 职工姓名: 李四 部门编号: 2
职工号: 3 职工姓名: 王五 部门编号: 3
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
请输入您的选择:
```

至此初始化数据功能完毕，测试代码可以注释或删除掉！

10、显示职工

功能描述：显示当前所有职工信息

10.1 显示职工函数声明

在workerManager.h中添加成员函数 `void Show_Emp();`

```
//显示职工
void Show_Emp();
```

10.2 显示职工函数实现

在workerManager.cpp中实现成员函数 `void Show_Emp();`

```
//显示职工
void WorkerManager::Show_Emp()
{
    if (this->m_FileIsEmpty)
    {
        cout << "文件不存在或记录为空！" << endl;
    }
    else
    {
        for (int i = 0; i < m_EmpNum; i++)
        {
            //利用多态调用接口
            this->m_EmpArray[i]->showInfo();
        }
    }

    system("pause");
    system("cls");
}
```

10.3 测试显示职工

在main函数分支 2 选项中，调用显示职工接口

```
switch (choice)
{
    case 0: //退出系统
        wm.exitSystem();
        break;
    case 1: //添加职工
        wm.Add_Emp();
        break;
    case 2: //显示职工
        wm.Show_Emp(); //添加显示职工接口
        break;
    case 3: //删除职工
        break;
    case 4: //修改职工
        break;
    case 5: //查找职工
        break;
    case 6: //排序职工
        break;
    case 7: //清空文件
        break;
    default:
        system("cls");
        break;
}
```

测试时分别测试 文件为空和文件不为空两种情况

测试效果：

测试1-文件不存在或者为空情况

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
** 欢迎使用职工管理系统! **
*****
0.退出管理程序
1.增加职工信息
2.显示职工信息
3.删除离职职工
4.修改职工信息
5.查找职工信息
6.按照编号排序
7.清空所有文档
*****  
请输入您的选择:  
2  
文件不存在或记录为空!  
请按任意键继续. . .
```

测试2 - 文件存在且有记录情况

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
** 欢迎使用职工管理系统! **
*****
0.退出管理程序
1.增加职工信息
2.显示职工信息
3.删除离职职工
4.修改职工信息
5.查找职工信息
6.按照编号排序
7.清空所有文档
*****  
请输入您的选择:  
2  
职工编号: 1 职工姓名: 张三 岗位: 员工 岗位职责: 完成经理交给的任务  
职工编号: 2 职工姓名: 李四 岗位: 经理 岗位职责: 完成老板交给的任务, 并下发任务给员工  
职工编号: 3 职工姓名: 王五 岗位: 总裁 岗位职责: 管理公司所有事务  
职工编号: 4 职工姓名: 赵六 岗位: 经理 岗位职责: 完成老板交给的任务, 并下发任务给员工  
请按任意键继续. . .
```

测试完毕，至此，显示所有职工信息功能实现

11、删除职工

功能描述：按照职工的编号进行删除职工操作

11.1 删除职工函数声明

在workerManager.h中添加成员函数 void Del_Emp();

```
//删除职工  
void Del_Emp();
```

11.2 职工是否存在函数声明

很多功能都需要用到根据职工是否存在来进行操作如：删除职工、修改职工、查找职工

因此添加该公告函数，以便后续调用

在workerManager.h中添加成员函数 int IsExist(int id);

```
//按照职工编号判断职工是否存在,若存在返回职工在数组中位置,不存在返回-1  
int IsExist(int id);
```

11.3 职工是否存在函数实现

在workerManager.cpp中实现成员函数 int IsExist(int id);

```
int WorkerManager::IsExist(int id)  
{  
    int index = -1;  
  
    for (int i = 0; i < this->m_EmpNum; i++)  
    {  
        if (this->m_EmpArray[i]->m_Id == id)  
        {  
            index = i;  
  
            break;  
        }  
    }  
  
    return index;  
}
```

11.4 删除职工函数实现

在workerManager.cpp中实现成员函数 void Del_Emp();

```

//删除职工
void WorkerManager::Del_Emp()
{
    if (this->m_FileIsEmpty)
    {
        cout << "文件不存在或记录为空! " << endl;
    }
    else
    {
        //按职工编号删除
        cout << "请输入想要删除的职工号: " << endl;
        int id = 0;
        cin >> id;

        int index = this->IsExist(id);

        if (index != -1) //说明index上位置数据需要删除
        {
            for (int i = index; i < this->m_EmpNum - 1; i++)
            {
                this->m_EmpArray[i] = this->m_EmpArray[i + 1];
            }
            this->m_EmpNum--;
        }

        this->save(); //删除后数据同步到文件中
        cout << "删除成功! " << endl;
    }
    else
    {
        cout << "删除失败, 未找到该职工" << endl;
    }
}

system("pause");
system("cls");
}

```

11.5 测试删除职工

在main函数分支 3 选项中，调用删除职工接口

```
switch (choice)
{
case 0: //退出系统
    wm.exitSystem();
    break;
case 1: //添加职工
    wm.Add_Emp();
    break;
case 2: //显示职工
    wm.Show_Emp();
    break;
case 3: //删除职工
    wm.Del_Emp();
    break;
case 4: //修改职工
    break;
case 5: //查找职工
    break;
case 6: //排序职工
    break;
case 7: //清空文件
    break;
default:
    system("cls");
    break;
}
```

调用删除职工接口

测试1 - 删除不存在职工情况

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****

请输入您的选择:
3
请输入想要删除的职工号:
1000
删除失败, 未找到该职工
请按任意键继续. . .
```

测试2 - 删除存在的职工情况

删除成功提示图:

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****
```

请输入您的选择:

3
请输入想要删除的职工号:

1

删除成功!

请按任意键继续. . . ■

再次显示所有职工信息, 确保已经删除

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****

请输入您的选择:
2
职工编号: 2 职工姓名: 李四 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
职工编号: 3 职工姓名: 王五 岗位: 总裁 岗位职责: 管理公司所有事务
职工编号: 4 职工姓名: 赵六 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
请按任意键继续. . .
```

查看文件中信息，再次核实员工已被完全删除

```
empFile.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
2 李四 2
3 王五 3
4 赵六 2
```

至此，删除职工功能实现完毕！

12、修改职工

功能描述：能够按照职工的编号对职工信息进行修改并保存

12.1 修改职工函数声明

在workerManager.h中添加成员函数 `void Mod_Emp();`

```
//修改职工  
void Mod_Emp();
```

12.2 修改职工函数实现

在workerManager.cpp中实现成员函数 `void Mod_Emp();`

```
//修改职工
void WorkerManager::Mod_Emp()
{
    if (this->m_FileIsEmpty)
    {
        cout << "文件不存在或记录为空！" << endl;
    }
    else
    {
        cout << "请输入修改职工的编号：" << endl;
        int id;
        cin >> id;

        int ret = this->IsExist(id);
        if (ret != -1)
        {
            //查找到编号的职工

            delete this->m_EmpArray[ret];

            int newId = 0;
            string newName = "";
            int dSelect = 0;

            cout << "查到：" << id << "号职工，请输入新职工号：" << endl;
            cin >> newId;

            cout << "请输入新姓名：" << endl;
            cin >> newName;

            cout << "请输入岗位：" << endl;
            cout << "1、普通职工" << endl;
            cout << "2、经理" << endl;
            cout << "3、老板" << endl;
            cin >> dSelect;

            Worker * worker = NULL;
            switch (dSelect)
            {
                case1:
                    worker = new Employee(newId, newName, dSelect);
                    break;
                case2:
                    worker = new Manager(newId, newName, dSelect);
                    break;
                case 3:
                    worker = new Boss(newId, newName, dSelect);
                    break;
                default:
                    break;
            }

            //更改数据 到数组中
            this->m_EmpArray[ret] = worker;
        }
    }
}
```

```

        cout << "修改成功!" << endl;

        //保存到文件中
        this->save();
    }
    else
    {
        cout << "修改失败, 查无此人" << endl;
    }
}

//按任意键 清屏
system("pause");
system("cls");
}

```

12.3 测试修改职工

在main函数分支 4 选项中，调用修改职工接口

```

switch (choice)
{
case 0: //退出系统
    wm.exitSystem();
    break;
case 1: //添加职工
    wm.Add_Emp();
    break;
case 2: //显示职工
    wm.Show_Emp();
    break;
case 3: //删除职工
    wm.Del_Emp();
    break;
case 4: //修改职工
    wm.Mod_Emp(); // 调用修改职工接口
    break;
case 5: //查找职工
    break;
case 6: //排序职工
    break;
case 7: //清空文件
    break;
default:
    system("cls");
    break;
}

```

测试1 - 修改不存在职工情况

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
****

请输入您的选择:
4
请输入修改职工的编号:
1000
修改失败, 查无此人
请按任意键继续. . .
```

测试2 - 修改存在职工情况, 例如将职工 "李四" 改为 "赵四"

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
****

请输入您的选择:
4
请输入修改职工的编号:
2
查到: 2号职工, 请输入新职工号:
2
请输入新姓名:
赵四
请输入岗位:
1、普通职工
2、经理
3、老板
2
修改成功
请按任意键继续. . .
```

修改后再次查看所有职工信息, 并确认修改成功

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0.退出管理程序 *****
***** 1.增加职工信息 *****
***** 2.显示职工信息 *****
***** 3.删除离职职工 *****
***** 4.修改职工信息 *****
***** 5.查找职工信息 *****
***** 6.按照编号排序 *****
***** 7.清空所有文档 *****
***** *****
请输入您的选择:
2
职工编号: 2 职工姓名: 赵四 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
职工编号: 3 职工姓名: 王五 岗位: 总裁 岗位职责: 管理公司所有事务
职工编号: 4 职工姓名: 赵六 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
请按任意键继续. . .
```

再次确认文件中信息也同步更新

```
empFile.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
2 赵四 2
3 王五 3
4 赵六 2
```

至此，修改职工功能已实现！

13、查找职工

功能描述：提供两种查找职工方式，一种按照职工编号，一种按照职工姓名

13.1 查找职工函数声明

在workerManager.h中添加成员函数 void Find_Emp();

```
//查找职工  
void Find_Emp();
```

13.2 查找职工函数实现

在workerManager.cpp中实现成员函数 void Find_Emp();

```
//查找职工
void WorkerManager::Find_Emp()
{
    if (this->m_FileIsEmpty)
    {
        cout << "文件不存在或记录为空! " << endl;
    }
    else
    {
        cout << "请输入查找的方式: " << endl;
        cout << "1、按职工编号查找" << endl;
        cout << "2、按姓名查找" << endl;

        int select = 0;
        cin >> select;

        if (select == 1) //按职工号查找
        {
            int id;
            cout << "请输入查找的职工编号: " << endl;
            cin >> id;

            int ret = IsExist(id);
            if (ret != -1)
            {
                cout << "查找成功! 该职工信息如下: " << endl;
                this->m_EmpArray[ret]->showInfo();
            }
            else
            {
                cout << "查找失败, 查无此人" << endl;
            }
        }
        else if(select == 2) //按姓名查找
        {
            string name;
            cout << "请输入查找的姓名: " << endl;
            cin >> name;

            bool flag = false; //查找到的标志
            for (int i = 0; i < m_EmpNum; i++)
            {
                if (m_EmpArray[i]->m_Name == name)
                {
                    cout << "查找成功, 职工编号为: "
                    << m_EmpArray[i]->m_Id
                    << " 号的信息如下: " << endl;

                    flag = true;
                    this->m_EmpArray[i]->showInfo();
                }
            }
            if (flag == false)
```

```

    {
        //查无此人
        cout << "查找失败，查无此人" << endl;
    }
} else
{
    cout << "输入选项有误" << endl;
}
}

system("pause");
system("cls");
}

```

13.3 测试查找职工

在main函数分支 5 选项中，调用查找职工接口

```

switch (choice)
{
case 0: //退出系统
    wm.exitSystem();
    break;
case 1: //添加职工
    wm.Add_Emp();
    break;
case 2: //显示职工
    wm.Show_Emp();
    break;
case 3: //删除职工
    wm.Del_Emp();
    break;
case 4: //修改职工
    wm.Mod_Emp();
    break;
case 5: //查找职工
    wm.Find_Emp();
    break; // 调用查找职工接口
case 6: //排序职工
    break;
case 7: //清空文件
    break;
default:
    system("cls");
    break;
}

```

测试1 - 按照职工编号查找 - 查找不存在职工

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
****

请输入您的选择:
5
请输入查找的方式:
1、按职工编号查找
2、按姓名查找
1
请输入查找的职工编号:
1000
查找失败，查无此人
请按任意键继续. . .
```

测试2 - 按照职工编号查找 - 查找存在职工

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
****

请输入您的选择:
5
请输入查找的方式:
1、按职工编号查找
2、按姓名查找
1
请输入查找的职工编号:
2
查找成功! 该职工信息如下:
职工编号: 2 职工姓名: 赵四 岗位: 经理 岗位职责: 完成老板交给的任务, 并下发任务给员工
请按任意键继续. . .
```

测试3 - 按照职工姓名查找 - 查找不存在职工

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****

请输入您的选择:
5
请输入查找的方式:
1、按职工编号查找
2、按姓名查找
2
请输入查找的姓名:
张三丰
查找失败, 查无此人
请按任意键继续. . .
```

测试4 - 按照职工姓名查找 - 查找存在职工 (如果出现重名, 也一并显示, 在文件中可以添加重名职工)

例如 添加两个王五的职工, 然后按照姓名查找王五

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****
```

请输入您的选择:

2

职工编号: 2 职工姓名: 赵四	岗位: 经理	岗位职责: 完成老板交给的任务, 并下发任务给员工
职工编号: 3 职工姓名: 王五	岗位: 总裁	岗位职责: 管理公司所有事务
职工编号: 4 职工姓名: 赵六	岗位: 经理	岗位职责: 完成老板交给的任务, 并下发任务给员工
职工编号: 1 职工姓名: 王五	岗位: 员工	岗位职责: 完成经理交给的任务

请按任意键继续. . .

↑

同名职工

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
*****

请输入您的选择:
5
请输入查找的方式:
1、按职工编号查找
2、按姓名查找
2
请输入查找的姓名:
王五
查找成功, 职工编号为: 3 号的信息如下:
职工编号: 3 职工姓名: 王五      岗位: 总裁      岗位职责: 管理公司所有事务
查找成功, 职工编号为: 1 号的信息如下:
职工编号: 1 职工姓名: 王五      岗位: 员工      岗位职责: 完成经理交给的任务
请按任意键继续. . .
```

至此，查找职工功能实现完毕！

14、排序

功能描述：按照职工编号进行排序，排序的顺序由用户指定

14.1 排序函数声明

在workerManager.h中添加成员函数 `void Sort_Emp();`

```
//排序职工
void Sort_Emp();
```

14.2 排序函数实现

在workerManager.cpp中实现成员函数 `void Sort_Emp();`

```

//排序职工
void WorkerManager::Sort_Emp()
{
    if (this->m_FileIsEmpty)
    {
        cout << "文件不存在或记录为空！" << endl;
        system("pause");
        system("cls");
    }
    else
    {
        cout << "请选择排序方式： " << endl;
        cout << "1、按职工号进行升序" << endl;
        cout << "2、按职工号进行降序" << endl;

        int select = 0;
        cin >> select;

        for (int i = 0; i < m_EmpNum; i++)
        {
            int minOrMax = i;
            for (int j = i + 1; j < m_EmpNum; j++)
            {
                if (select == 1) //升序
                {
                    if (m_EmpArray[minOrMax]->m_Id > m_EmpArray[j]->m_Id)
                    {
                        minOrMax = j;
                    }
                }
                else //降序
                {
                    if (m_EmpArray[minOrMax]->m_Id < m_EmpArray[j]->m_Id)
                    {
                        minOrMax = j;
                    }
                }
            }

            if (i != minOrMax)
            {
                Worker * temp = m_EmpArray[i];
                m_EmpArray[i] = m_EmpArray[minOrMax];
                m_EmpArray[minOrMax] = temp;
            }
        }

        cout << "排序成功，排序后结果为：" << endl;
        this->save();
        this->Show_Emp();
    }
}

```

}

14.3 测试排序功能

在main函数分支 6 选项中，调用排序职工接口

```
switch (choice)
{
    case 0: //退出系统
        wm.exitSystem();
        break;
    case 1: //添加职工
        wm.Add_Emp();
        break;
    case 2: //显示职工
        wm.Show_Emp();
        break;
    case 3: //删除职工
        wm.Del_Emp();
        break;
    case 4: //修改职工
        wm.Mod_Emp();
        break;
    case 5: //查找职工
        wm.Find_Emp();
        break;
    case 6: //排序职工
        wm.Sort_Emp();
        break;
    case 7: //清空文件
        break;
    default:
        system("cls");
        break;
}
```

调用排序职工接口

测试：

首先我们添加一些职工，序号是无序的，例如：

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0.退出管理程序 *****
***** 1.增加职工信息 *****
***** 2.显示职工信息 *****
***** 3.删除离职职工 *****
***** 4.修改职工信息 *****
***** 5.查找职工信息 *****
***** 6.按照编号排序 *****
***** 7.清空所有文档 *****
****

请输入您的选择:
2
职工编号: 2 职工姓名: 赵四 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
职工编号: 3 职工姓名: 王五 岗位: 总裁 岗位职责: 管理公司所有事务
职工编号: 4 职工姓名: 赵六 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
职工编号: 1 职工姓名: 王五 岗位: 员工 岗位职责: 完成经理交给的任务
职工编号: 5 职工姓名: 孙悟空 岗位: 高工 岗位职责: 完成经理交给的任务
职工编号: 6 职工姓名: 唐三藏 岗位: 总裁 岗位职责: 管理公司所有事务
请按任意键继续. . .
```

测试 - 升序排序

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0.退出管理程序 *****
***** 1.增加职工信息 *****
***** 2.显示职工信息 *****
***** 3.删除离职职工 *****
***** 4.修改职工信息 *****
***** 5.查找职工信息 *****
***** 6.按照编号排序 *****
***** 7.清空所有文档 *****
****

请输入您的选择:
6
请选择排序方式:
1、按职工号进行升序
2、按职工号进行降序
1
排序成功,排序后结果为:
职工编号: 1 职工姓名: 王五 岗位: 员工 岗位职责: 完成经理交给的任务
职工编号: 2 职工姓名: 赵四 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
职工编号: 3 职工姓名: 王五 岗位: 总裁 岗位职责: 管理公司所有事务
职工编号: 4 职工姓名: 赵六 岗位: 经理 岗位职责: 完成老板交给的任务,并下发任务给员工
职工编号: 5 职工姓名: 孙悟空 岗位: 员工 岗位职责: 完成经理交给的任务
职工编号: 6 职工姓名: 唐三藏 岗位: 总裁 岗位职责: 管理公司所有事务
请按任意键继续. . .
```

文件同步更新

empFile.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
1 王五 1
2 赵四 2
3 王五 3
4 赵六 2
5 孙悟空 1
6 唐三藏 3
```

测试 - 降序排序

F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe

```
***** 欢迎使用职工管理系统! *****
***** 0. 退出管理程序 *****
***** 1. 增加职工信息 *****
***** 2. 显示职工信息 *****
***** 3. 删除离职职工 *****
***** 4. 修改职工信息 *****
***** 5. 查找职工信息 *****
***** 6. 按照编号排序 *****
***** 7. 清空所有文档 *****
***** ***** ***** ***** ***** ***** ***** *****
```

请输入您的选择:

6

请选择排序方式:

1、按职工号进行升序
2、按职工号进行降序

2

排序成功, 排序后结果为:

职工编号: 6	职工姓名: 唐三藏	岗位: 总裁	岗位职责: 管理公司所有事务
职工编号: 5	职工姓名: 孙悟空	岗位: 员工	岗位职责: 完成经理交给的任务
职工编号: 4	职工姓名: 赵六	岗位: 经理	岗位职责: 完成老板交给的任务, 并下发任务给员工
职工编号: 3	职工姓名: 王五	岗位: 总裁	岗位职责: 管理公司所有事务
职工编号: 2	职工姓名: 赵四	岗位: 经理	岗位职责: 完成老板交给的任务, 并下发任务给员工
职工编号: 1	职工姓名: 王五	岗位: 员工	岗位职责: 完成经理交给的任务

按任意键继续. . .

文件同步更新



编号	姓名	年龄
6	唐三藏	3
5	孙悟空	1
4	赵六	2
3	王五	3
2	赵四	2
1	王五	1

至此，职工按照编号排序的功能实现完毕！

15、清空文件

功能描述：将文件中记录数据清空

15.1 清空函数声明

在workerManager.h中添加成员函数 `void Clean_File();`

```
//清空文件  
void Clean_File();
```

15.2 清空函数实现

在workerManager.cpp中实现员函数 `void Clean_File();`

```

//清空文件
void WorkerManager::Clean_File()
{
    cout << "确认清空? " << endl;
    cout << "1、确认" << endl;
    cout << "2、返回" << endl;

    int select = 0;
    cin >> select;

    if (select == 1)
    {
        //打开模式 ios::trunc 如果存在删除文件并重新创建
        ofstream ofs(FILENAME, ios::trunc);
        ofs.close();

        if (this->m_EmpArray != NULL)
        {
            for (int i = 0; i < this->m_EmpNum; i++)
            {
                if (this->m_EmpArray[i] != NULL)
                {
                    delete this->m_EmpArray[i];
                }
            }
            this->m_EmpNum = 0;
            delete[] this->m_EmpArray;
            this->m_EmpArray = NULL;
            this->m_FileIsEmpty = true;
        }
        cout << "清空成功! " << endl;
    }

    system("pause");
    system("cls");
}

```

15.3 测试清空文件

在main函数分支 7 选项中，调用清空文件接口

```
switch (choice)
{
    case 0: //退出系统
        wm.exitSystem();
        break;
    case 1: //添加职工
        wm.Add_Emp();
        break;
    case 2: //显示职工
        wm.Show_Emp();
        break;
    case 3: //删除职工
        wm.Del_Emp();
        break;
    case 4: //修改职工
        wm.Mod_Emp();
        break;
    case 5: //查找职工
        wm.Find_Emp();
        break;
    case 6: //排序职工
        wm.Sort_Emp();
        break;
    case 7: //清空文件
        wm.Clean_File();
        break;
    default:
        system("cls");
        break;
}
```

调用清空文件接口

测试：确认清空文件

```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0.退出管理程序 *****
***** 1.增加职工信息 *****
***** 2.显示职工信息 *****
***** 3.删除离职职工 *****
***** 4.修改职工信息 *****
***** 5.查找职工信息 *****
***** 6.按照编号排序 *****
***** 7.清空所有文档 *****
*****

请输入您的选择:
7
确认清空?
1、确认
2、返回
1
清空成功!
请按任意键继续. . .

```

再次查看文件中数据，记录已为空

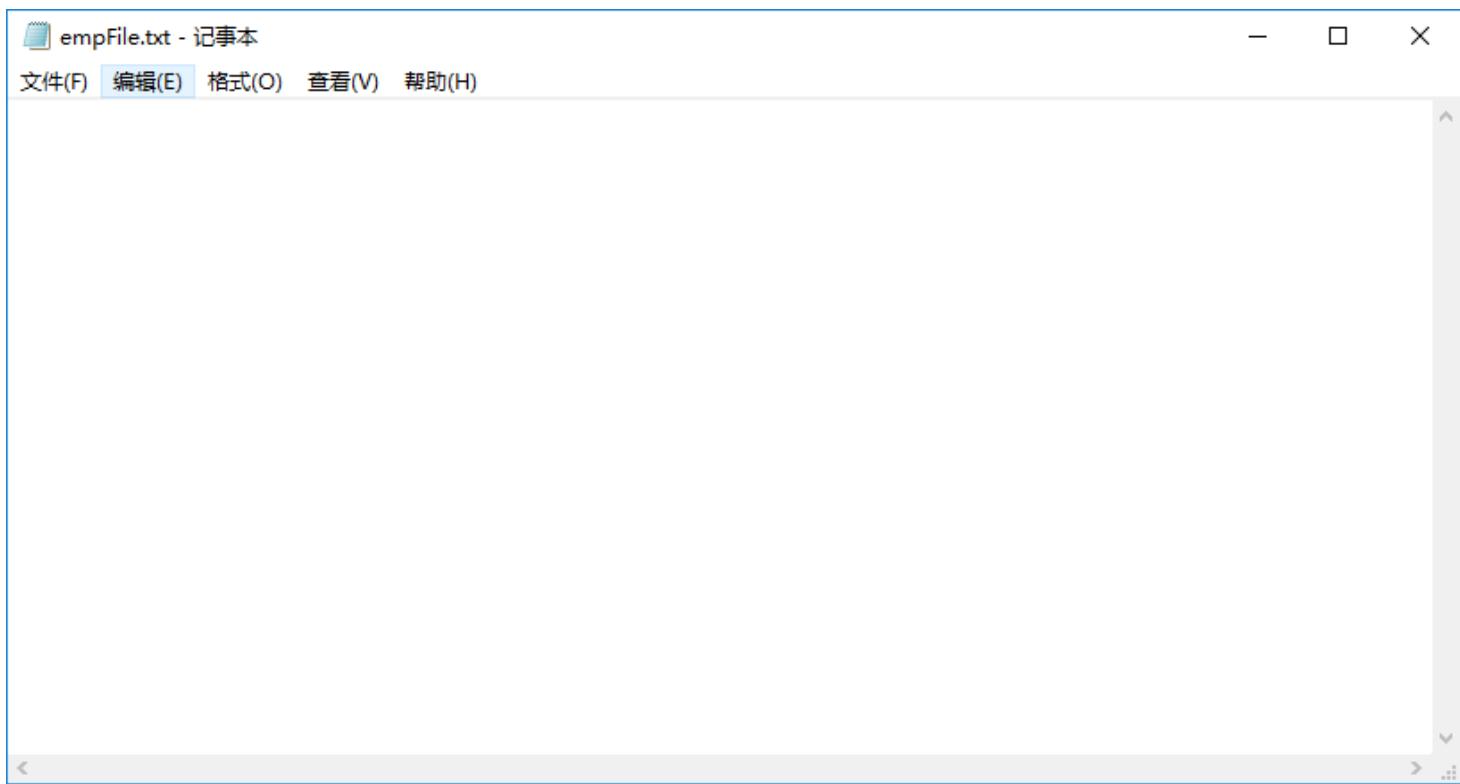
```
F:\VS项目\基于多态的职工管理系统\Debug\基于多态的职工管理系统.exe
*****
***** 欢迎使用职工管理系统! *****
***** 0.退出管理程序 *****
***** 1.增加职工信息 *****
***** 2.显示职工信息 *****
***** 3.删除离职职工 *****
***** 4.修改职工信息 *****
***** 5.查找职工信息 *****
***** 6.按照编号排序 *****
***** 7.清空所有文档 *****
*****



请输入您的选择:
2
文件不存在或记录为空!
请按任意键继续. . .

```

打开文件，里面数据已确保清空，该功能需要慎用！



随着清空文件功能实现，本案例制作完毕 ^_^

演讲比赛流程管理系统

1、演讲比赛程序需求



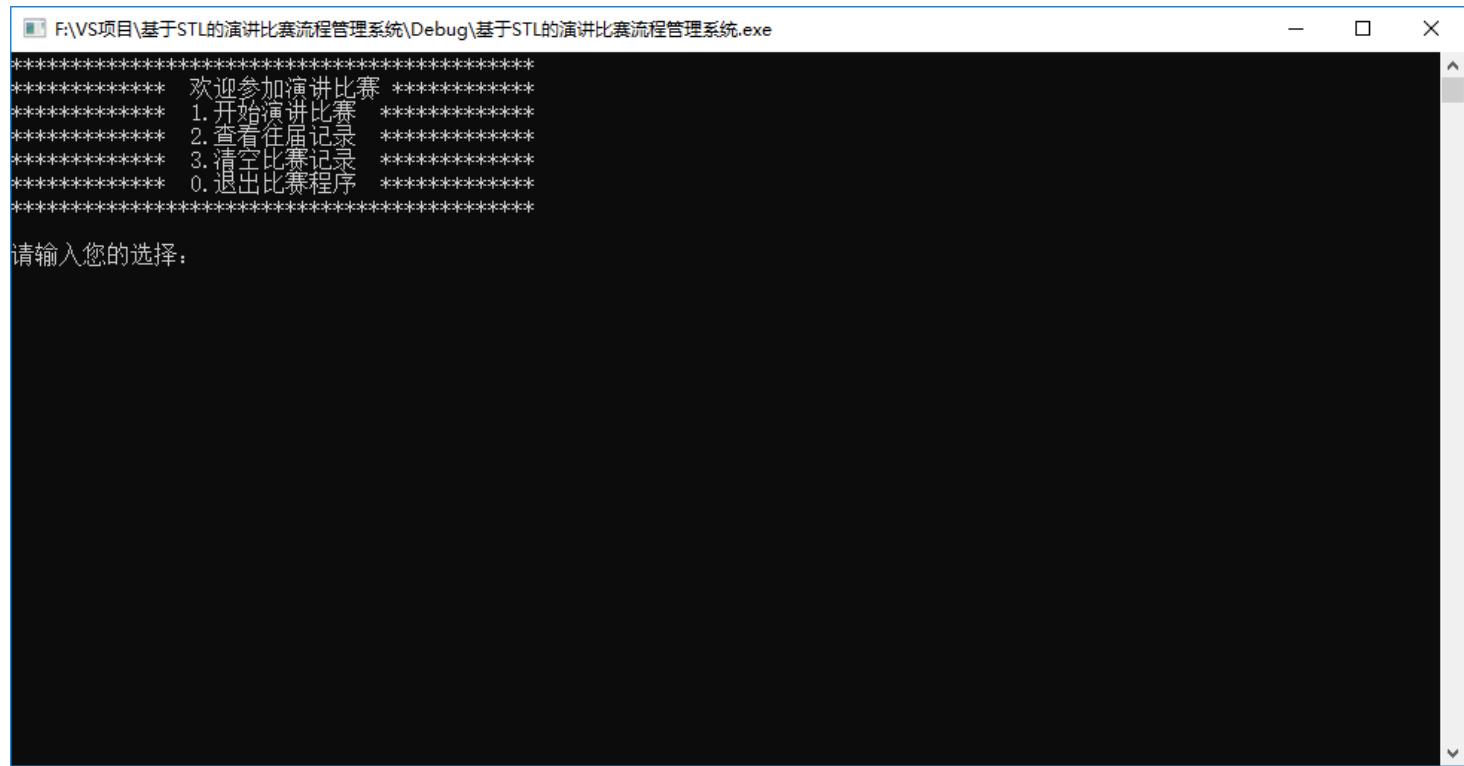
1.1 比赛规则

- 学校举行一场演讲比赛，共有**12个人**参加。**比赛共两轮**，第一轮为淘汰赛，第二轮为决赛。
- 比赛方式：**分组比赛，每组6个人**；选手每次要随机分组，进行比赛
- 每名选手都有对应的**编号**，如 10001 ~ 10012
- 第一轮分为两个小组，每组6个人。整体按照选手编号进行**抽签**后顺序演讲。
- 当小组演讲完后，淘汰组内排名最后的三个选手，**前三名晋级**，进入下一轮的比赛。
- 第二轮为决赛，**前三名胜出**
- 每轮比赛过后需要**显示晋级选手的信息**

1.2 程序功能

- 开始演讲比赛：完成整届比赛的流程，每个比赛阶段需要给用户一个提示，用户按任意键后继续下一个阶段
- 查看往届记录：查看之前比赛前三名结果，每次比赛都会记录到文件中，文件用.csv后缀名保存
- 清空比赛记录：将文件中数据清空
- 退出比赛程序：可以退出当前程序

1.3 程序效果图：



```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
*****
***** 欢迎参加演讲比赛 *****
***** 1. 开始演讲比赛 *****
***** 2. 查看往届记录 *****
***** 3. 清空比赛记录 *****
***** 0. 退出比赛程序 *****
*****
请输入您的选择:
```

2、项目创建

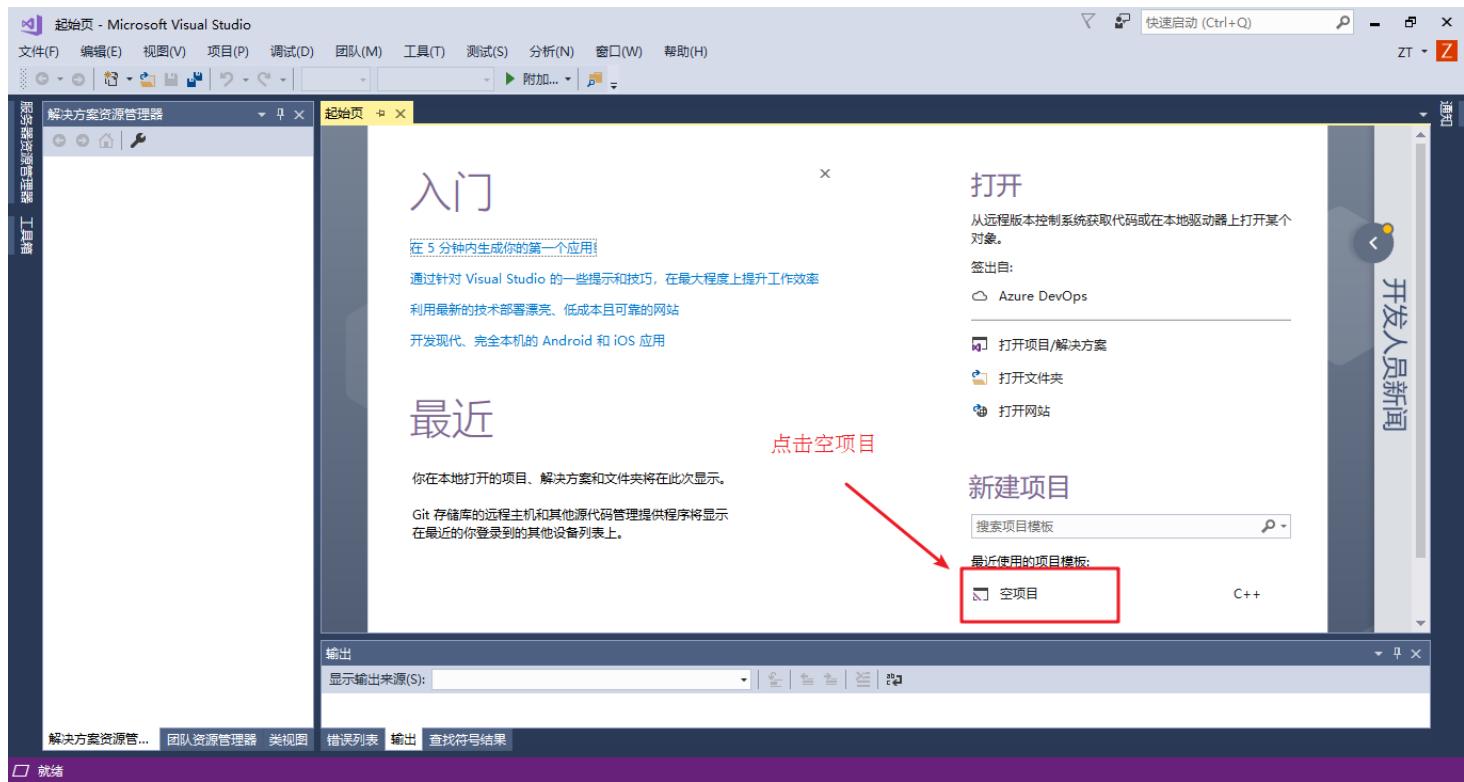
创建项目步骤如下：

- 创建新项目
- 添加文件

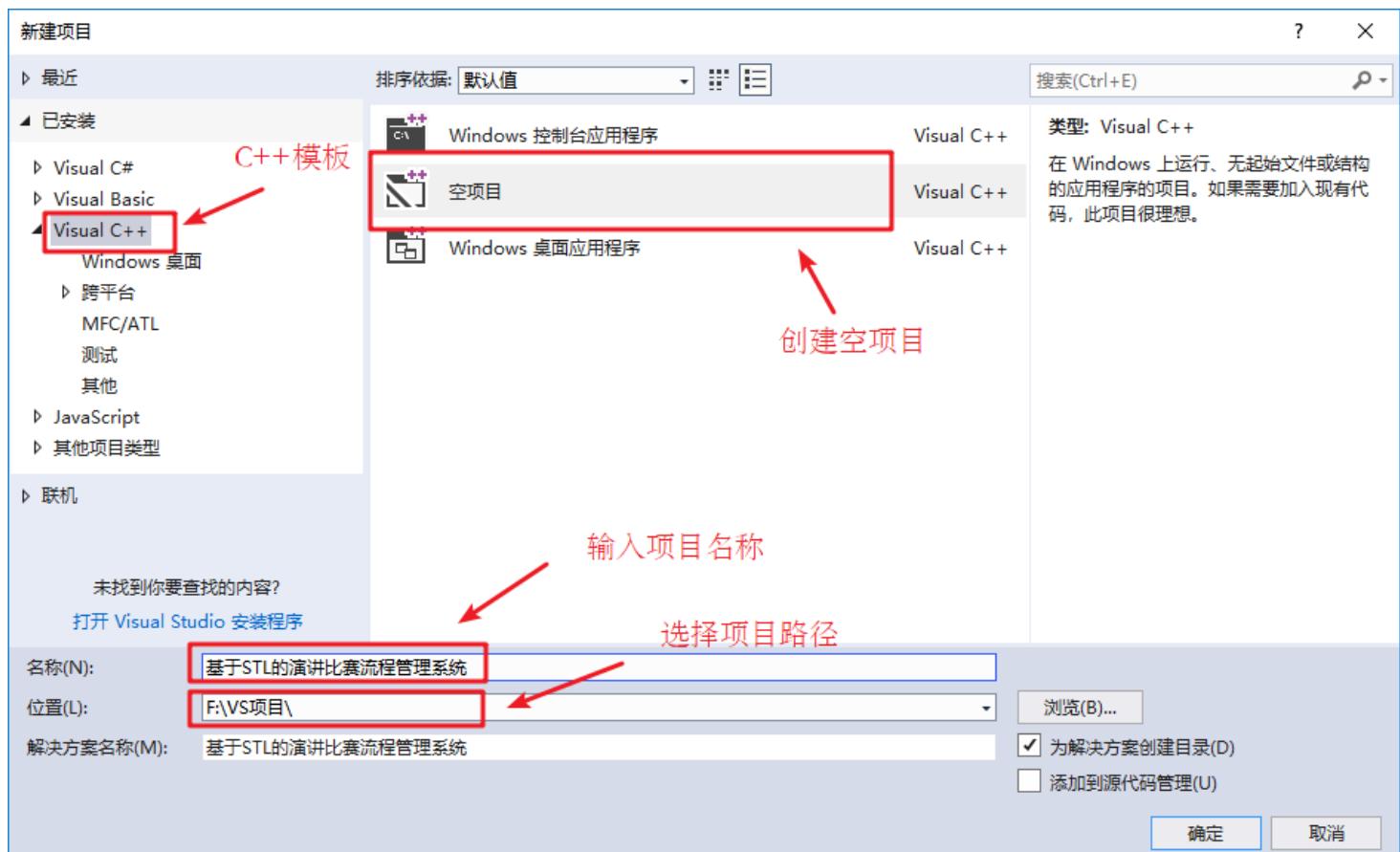
2.1 创建项目

- 打开vs2017后，点击创建新项目，创建新的C++项目

如图：

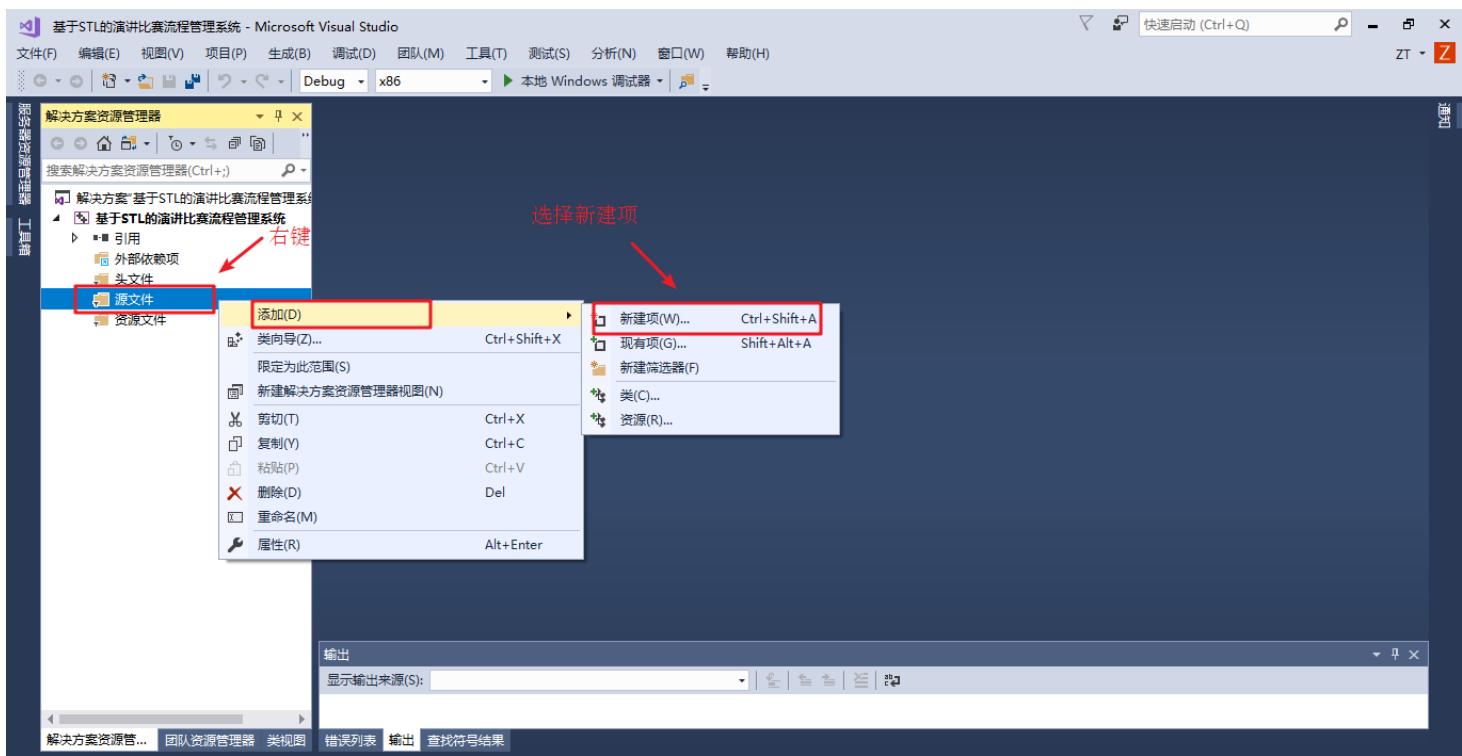


- 填写项目名称以及选取项目路径，点击确定生成项目



2.2 添加文件

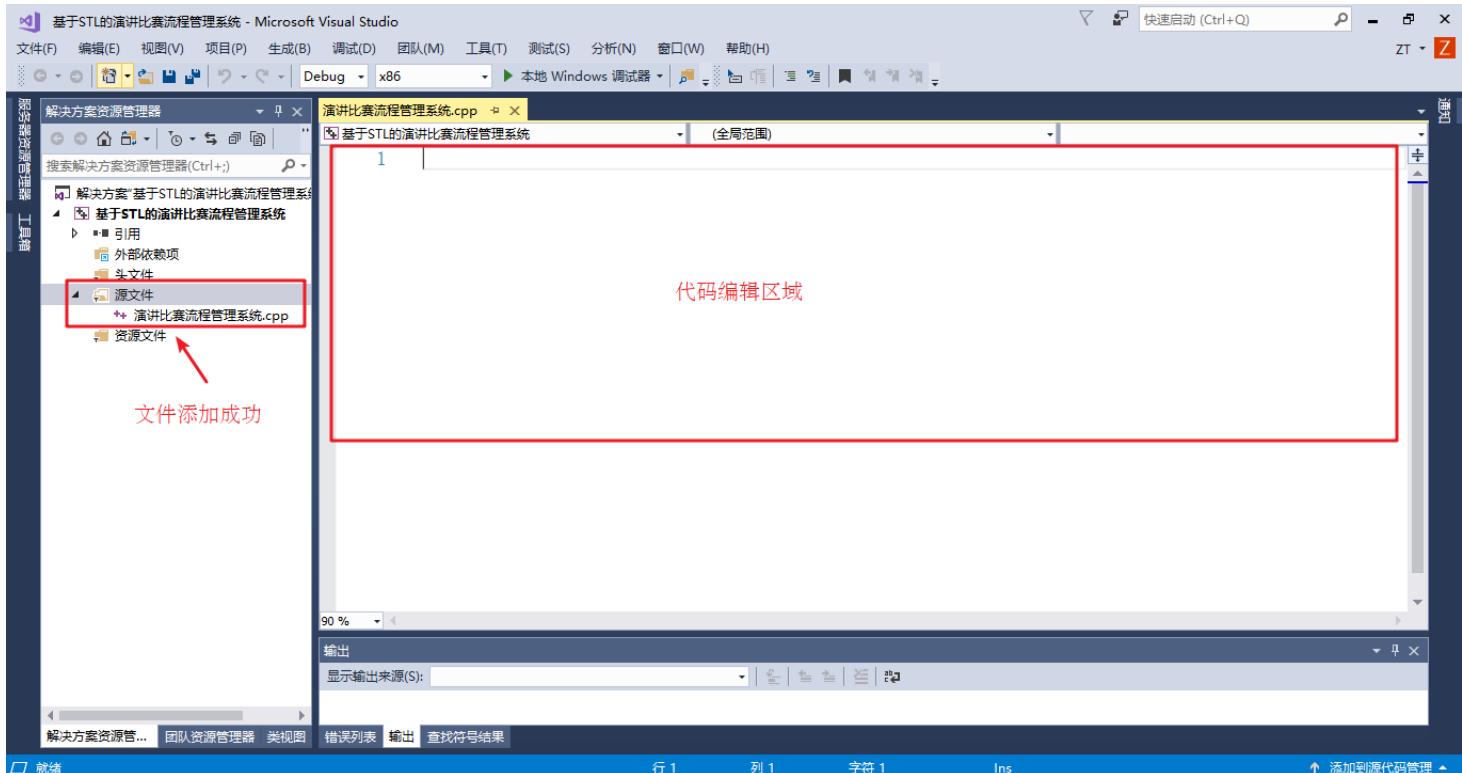
- 右键源文件，进行添加文件操作



- 填写文件名称，点击添加



- 生成文件成功，效果如下图



- 至此，项目已创建完毕

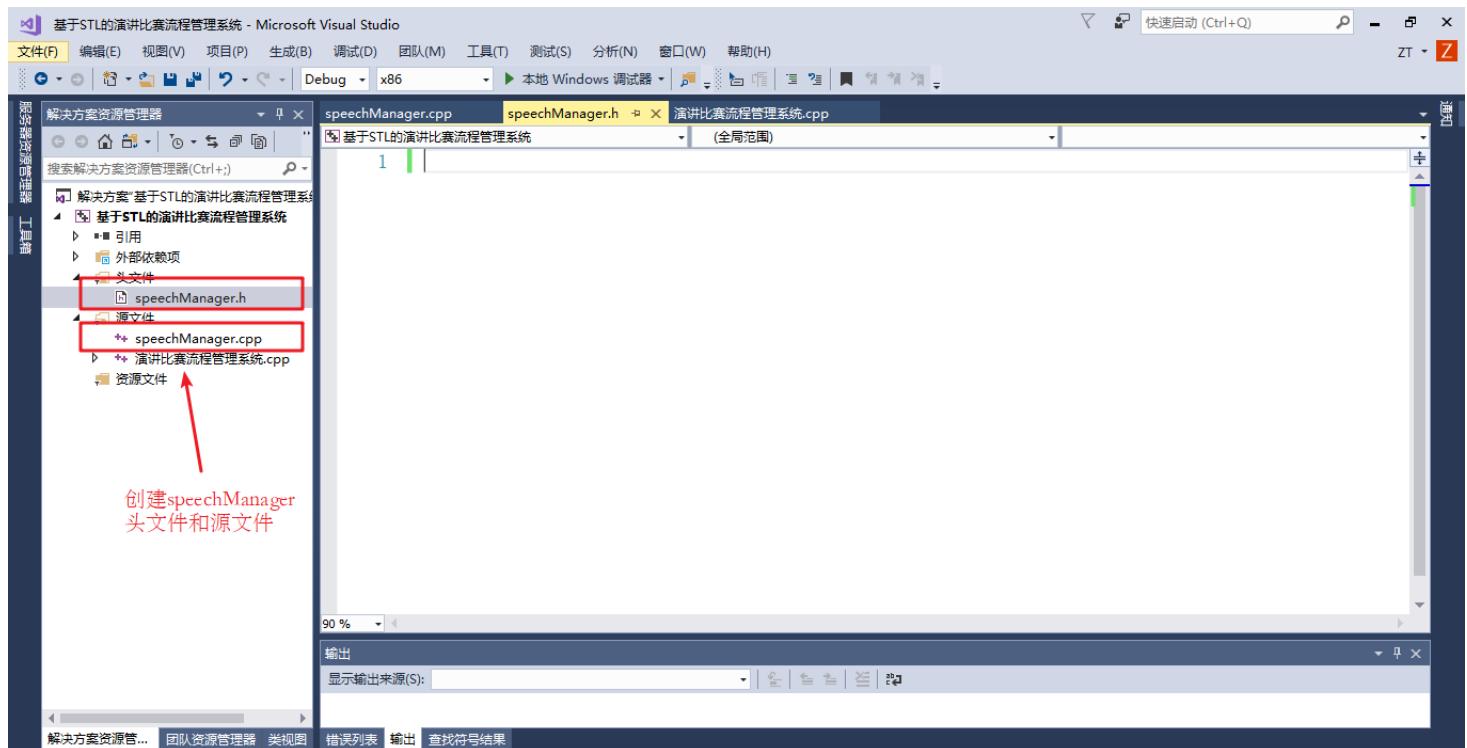
3、 创建管理类

功能描述：

- 提供菜单界面与用户交互
- 对演讲比赛流程进行控制
- 与文件的读写交互

3.1 创建文件

- 在头文件和源文件的文件夹下分别创建speechManager.h 和 speechManager.cpp文件



3.2 头文件实现

在speechManager.h中设计管理类

代码如下：

```
#pragma once
#include<iostream>
using namespace std;

//演讲管理类
class SpeechManager
{
public:

    //构造函数
    SpeechManager();

    //析构函数
    ~SpeechManager();
};

};
```

3.3 源文件实现

在speechManager.cpp中将构造和析构函数空实现补全

```
#include "speechManager.h"

SpeechManager::SpeechManager()
{
}

SpeechManager::~SpeechManager()
{
}
```

- 至此演讲管理类以创建完毕

4、 菜单功能

功能描述：与用户的沟通界面

4.1 添加成员函数

在管理类speechManager.h中添加成员函数 void show_Menu();

```
4 //演讲管理类
5 class SpeechManager
6 {
7     public:
8
9         //构造函数
10    SpeechManager();
11
12
13    //展示菜单
14    void show_Menu();
15
16
17    //析构函数
18    ~SpeechManager();
19
20};
```



4.2 菜单功能实现

- 在管理类speechManager.cpp中实现 show_Menu()函数

```
void SpeechManager::show_Menu()
{
    cout << "*****" << endl;
    cout << "***** 欢迎参加演讲比赛 *****" << endl;
    cout << "***** 1.开始演讲比赛 *****" << endl;
    cout << "***** 2.查看往届记录 *****" << endl;
    cout << "***** 3.清空比赛记录 *****" << endl;
    cout << "***** 0.退出比赛程序 *****" << endl;
    cout << "*****" << endl;
}
```

4.3 测试菜单功能

- 在演讲比赛流程管理系统.cpp中测试菜单功能

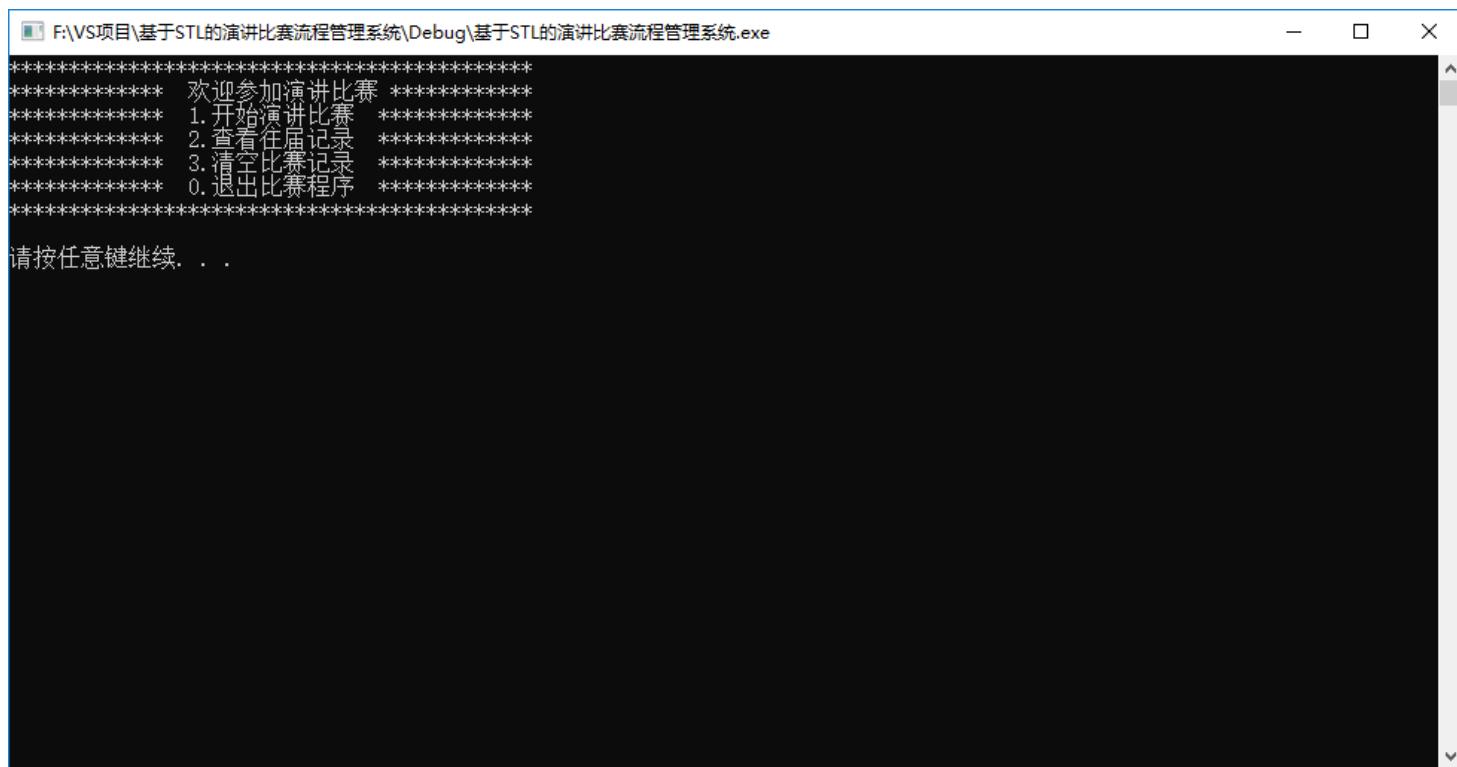
代码：

```
#include<iostream>
using namespace std;
#include "speechManager.h"

int main() {
    SpeechManager sm;

    sm.show_Menu();
    system("pause");
    return 0;
}
```

- 运行效果如图：



- 菜单界面搭建完毕

5、退出功能

5.1 提供功能接口

- 在main函数中提供分支选择，提供每个功能接口

代码：

```

int main() {
    SpeechManager sm;

    int choice = 0; //用来存储用户的选项

    while (true)
    {
        sm.show_Menu();

        cout << "请输入您的选择: " << endl;
        cin >> choice; // 接受用户的选项

        switch (choice)
        {
            case 1: //开始比赛
                break;
            case 2: //查看记录
                break;
            case 3: //清空记录
                break;
            case 0: //退出系统
                break;
            default:
                system("cls"); //清屏
                break;
        }
    }

    system("pause");
}

return 0;
}

```

5.2 实现退出功能

在speechManager.h中提供退出系统的成员函数 void exitSystem();

在speechManager.cpp中提供具体的功能实现

```

void SpeechManager::exitSystem()
{
    cout << "欢迎下次使用" << endl;
    system("pause");
    exit(0);
}

```

5.3 测试功能

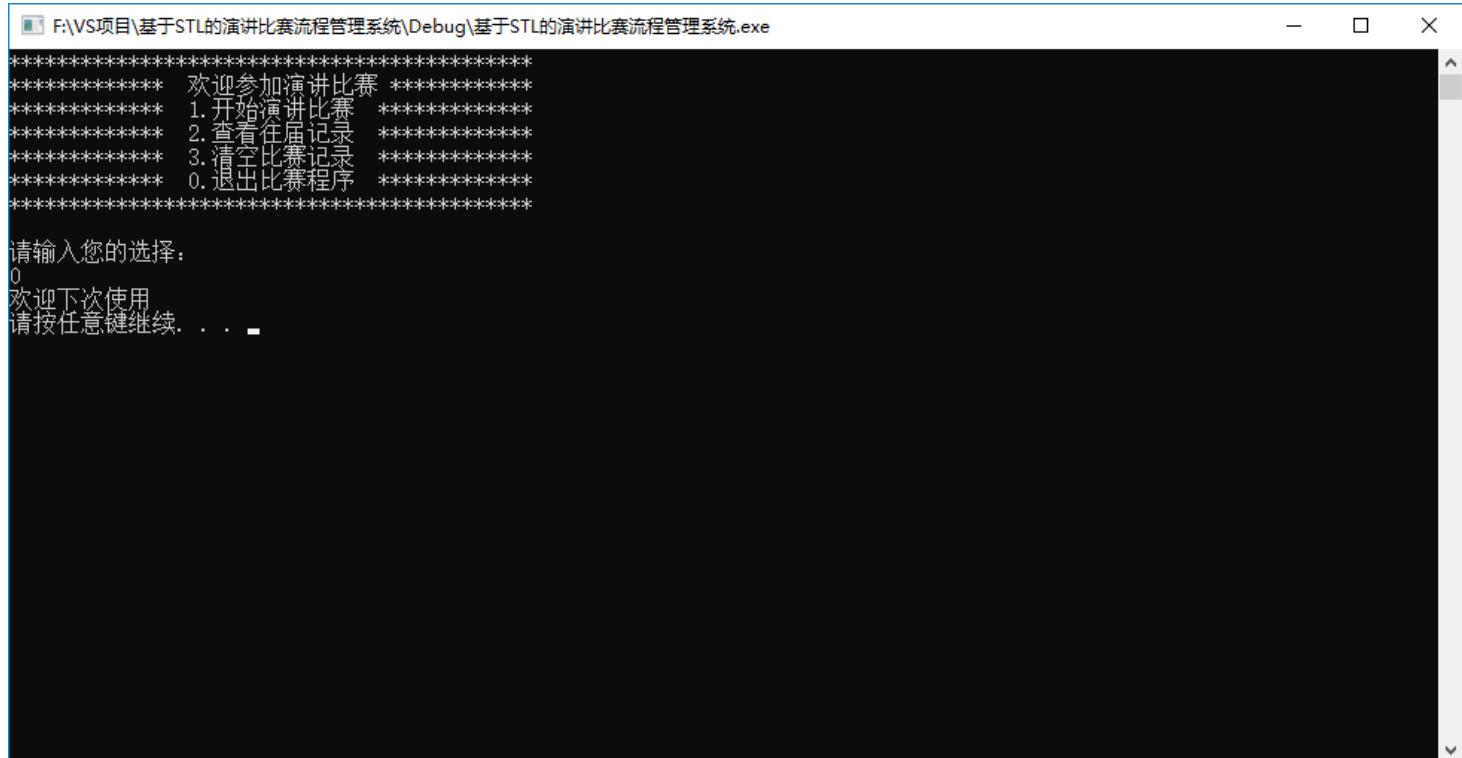
在main函数分支 0 选项中，调用退出程序的接口

```
while (true)
{
    sm.show_Menu();

    cout << "请输入您的选择: " << endl;
    cin >> choice; // 接受用户的选项

    switch (choice)
    {
        case 1: //开始比赛
            break;
        case 2: //查看记录
            break;
        case 3: //清空记录
            break;
        case 0: //退出系统
            sm.exitSystem();
            break;
        default:
            system("cls"); //清屏
            break;
    }
}
```

运行测试效果如图：



6、演讲比赛功能

6.1 功能分析

比赛流程分析：

抽签 → 开始演讲比赛 → 显示第一轮比赛结果 →

抽签 → 开始演讲比赛 → 显示前三名结果 → 保存分数

6.2 创建选手类

- 选手类中的属性包含：选手姓名、分数
- 头文件中创建 speaker.h文件，并添加代码：

```
#pragma once
#include<iostream>
using namespace std;

class Speaker
{
public:
    string m_Name; //姓名
    double m_Score[2]; //分数 最多有两轮得分
};
```

6.3 比赛

6.3.1 成员属性添加

- 在speechManager.h中添加属性

```
//比赛选手 容器 12人
vector<int>v1;

//第一轮晋级容器 6人
vector<int>v2;

//胜利前三名容器 3人
vector<int>vVictory;

//存放编号 以及对应的 具体选手 容器
map<int, Speaker> m_Speaker;
```

6.3.2 初始化属性

- 在speechManager.h中提供开始比赛的成员函数 void initSpeech();

```
//初始化属性
```

```
void initSpeech();
```

- 在speechManager.cpp中实现 void initSpeech();

```
void SpeechManager::initSpeech()
{
    //容器保证为空
    this->v1.clear();
    this->v2.clear();
    this->vVictory.clear();
    this->m_Speaker.clear();
    //初始化比赛轮数
    this->m_Index = 1;
}
```

- SpeechManager构造函数中调用 void initSpeech();

```
SpeechManager::SpeechManager()
{
    //初始化属性
    this->initSpeech();
}
```

6.3.3 创建选手

- 在speechManager.h中提供开始比赛的成员函数 void createSpeaker();

```
//初始化创建12名选手
```

```
void createSpeaker();
```

- 在speechManager.cpp中实现 void createSpeaker();

```

void SpeechManager::createSpeaker()
{
    string nameSeed = "ABCDEFGHIJKLM";
    for (int i = 0; i < nameSeed.size(); i++)
    {
        string name = "选手";
        name += nameSeed[i];

        Speaker sp;
        sp.m_Name = name;
        for (int i = 0; i < 2; i++)
        {
            sp.m_Score[i] = 0;
        }

        //12名选手编号
        this->v1.push_back(i + 10001);

        //选手编号 以及对应的选手 存放到map容器中
        this->m_Speaker.insert(make_pair(i + 10001, sp));
    }
}

```

- SpeechManager类的 构造函数中调用 void createSpeaker();

```

SpeechManager::SpeechManager()
{
    //初始化属性
    this->initSpeech();

    //创建选手
    this->createSpeaker();
}

```

- 测试 在main函数中，可以在创建完管理对象后，使用下列代码测试12名选手初始状态

```

for (map<int, Speaker>::iterator it = sm.m_Speaker.begin(); it != sm.m_Speaker.end(); it++)
{
    cout << "选手编号: " << it->first
    << " 姓名: " << it->second.m_Name
    << " 成绩: " << it->second.m_Score[0] << endl;
}

```

```
#include<iostream>
using namespace std;
#include "speechManager.h"
#include <string>

int main() {
    SpeechManager sm;

    //测试代码
    for (map<int, Speaker>::iterator it = sm.m_Speaker.begin(); it != sm.m_Speaker.end(); {
        cout << "选手编号: " << it->first
            << " 姓名: " << it->second.m_Name
            << " 成绩: " << it->second.m_Score[0] << endl;
    }

    int choice = 0; //用来存储用户的选项

    while (true)
    {
        sm.show_Menu();

        cout << "请输入您的选择: " << endl;
        cin >> choice; // 接受用户的选项

        switch (choice)
        {
            case 1: //开始比赛
                break;
            case 2: //查看记录
                break;
        }
    }
}
```

- 测试效果如图：

```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
选手编号: 10001 姓名: 选手A 成绩: 0
选手编号: 10002 姓名: 选手B 成绩: 0
选手编号: 10003 姓名: 选手C 成绩: 0
选手编号: 10004 姓名: 选手D 成绩: 0
选手编号: 10005 姓名: 选手E 成绩: 0
选手编号: 10006 姓名: 选手F 成绩: 0
选手编号: 10007 姓名: 选手G 成绩: 0
选手编号: 10008 姓名: 选手H 成绩: 0
选手编号: 10009 姓名: 选手I 成绩: 0
选手编号: 10010 姓名: 选手J 成绩: 0
选手编号: 10011 姓名: 选手K 成绩: 0
选手编号: 10012 姓名: 选手L 成绩: 0
*****
***** 欢迎参加演讲比赛 *****
***** 1. 开始演讲比赛 *****
***** 2. 查看往届记录 *****
***** 3. 清空比赛记录 *****
***** 0. 退出比赛程序 *****
*****
请输入您的选择:
```

- 测试完毕后，可以将测试代码删除或注释。

6.3.4 开始比赛成员函数添加

- 在speechManager.h中提供开始比赛的成员函数 `void startSpeech();`
- 该函数功能是主要控制比赛的流程

```
//开始比赛 - 比赛流程控制
void startSpeech();
```

- 在speechManager.cpp中将startSpeech的空实现先写入
- 我们可以先将整个比赛的流程 写到函数中

```
//开始比赛
void SpeechManager::startSpeech()
{
    //第一轮比赛
    //1、抽签

    //2、比赛

    //3、显示晋级结果

    //第二轮比赛

    //1、抽签

    //2、比赛

    //3、显示最终结果

    //4、保存分数
}
```

6.3.5 抽签

功能描述：

- 正式比赛前，所有选手的比赛顺序需要打乱，我们只需要将存放选手编号的容器 打乱次序即可
- 在speechManager.h中提供抽签的成员函数 **void speechDraw();**

```
//抽签
void speechDraw();
```

- 在speechManager.cpp中实现成员函数 **void speechDraw();**

```

void SpeechManager::speechDraw()
{
    cout << "第 " << this->m_Index << " >> 轮比赛选手正在抽签" << endl;
    cout << "-----" << endl;
    cout << "抽签后演讲顺序如下：" << endl;
    if (this->m_Index == 1)
    {
        random_shuffle(v1.begin(), v1.end());
        for (vector<int>::iterator it = v1.begin(); it != v1.end(); it++)
        {
            cout << *it << " ";
        }
        cout << endl;
    }
    else
    {
        random_shuffle(v2.begin(), v2.end());
        for (vector<int>::iterator it = v2.begin(); it != v2.end(); it++)
        {
            cout << *it << " ";
        }
        cout << endl;
    }
    cout << "-----" << endl;
    system("pause");
    cout << endl;
}

```

- 在startSpeech比赛流程控制的函数中，调用抽签函数

```

void SpeechManager::startSpeech()
{
    //第一轮比赛
    //1、抽签
    speechDraw();
    //2、比赛

    //3、显示晋级结果

    //第二轮比赛

    //1、抽签

    //2、比赛

    //3、显示最终结果

    //4、保存分数
}

```

- 在main函数中，分支1选项中，调用开始比赛的接口

```
while (true)
{
    sm.show_Menu();

    cout << "请输入您的选择: " << endl;
    cin >> choice; // 接受用户的选项

    switch (choice)
    {
        case 1: //开始比赛
            sm.startSpeech();
            break;
        case 2: //查看记录
            break;
        case 3: //清空记录
            break;
        case 0: //退出系统
            sm.exitSystem();
            break;
        default:
            system("cls"); //清屏
            break;
    }
}
```

- 测试

```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
*****
***** 欢迎参加演讲比赛 *****
***** 1. 开始演讲比赛 *****
***** 2. 查看往届记录 *****
***** 3. 清空比赛记录 *****
***** 0. 退出比赛程序 *****
*****

请输入您的选择:
1
第 << 1 >> 轮比赛选手正在抽签
抽签后演讲顺序如下:
10005 10008 10010 10012 10001 10003 10009 10004 10007 10002 10006 10011
请按任意键继续. . .
```

6.3.6 开始比赛

- 在speechManager.h中提供比赛的成员函数 void speechContest();

```
//比赛  
void speechContest();
```

- 在speechManager.cpp中实现成员函数 void speechContest();

```

void SpeechManager::speechContest()
{
    cout << "----- 第" << this->m_Index << "轮正式比赛开始: ----- " << endl;

    multimap<double, int, greater<int>> groupScore; //临时容器，保存key分数 value 选手编号

    int num = 0; //记录人员数，6个为1组

    vector <int> v_Src; //比赛的人员容器
    if (this->m_Index == 1)
    {
        v_Src = v1;
    }
    else
    {
        v_Src = v2;
    }

    //遍历所有参赛选手
    for (vector<int>::iterator it = v_Src.begin(); it != v_Src.end(); it++)
    {
        num++;

        //评委打分
        deque<double> d;
        for (int i = 0; i < 10; i++)
        {
            double score = (rand() % 401 + 600) / 10.f; // 600 ~ 1000
            //cout << score << " ";
            d.push_back(score);
        }

        sort(d.begin(), d.end(), greater<double>()); //排序
        d.pop_front();
        d.pop_back();

        double sum = accumulate(d.begin(), d.end(), 0.0f);
        double avg = sum / (double)d.size();

        //每个人平均分
        //cout << "编号: " << *it << " 选手: " << this->m_Speaker[*it].m_Name << " 获得
        this->m_Speaker[*it].m_Score[this->m_Index - 1] = avg;

        //6个人一组，用临时容器保存
        groupScore.insert(make_pair(avg, *it));
        if (num % 6 == 0)
        {

            cout << "第" << num / 6 << "小组比赛名次: " << endl;
            for (multimap<double, int, greater<int>>::iterator it = groupScore.begin()
            {
                cout << "编号: " << it->second << " 姓名: " << this->m_Speaker[:
            }

            int count = 0;
        }
    }
}

```

```

        //取前三名
        for (multimap<double, int, greater<int>>::iterator it = groupScore.begin();
        {
            if (this->m_Index == 1)
            {
                v2.push_back((*it).second);
            }
            else
            {
                vVictory.push_back((*it).second);
            }
        }

        groupScore.clear();

        cout << endl;

    }

}

cout << "----- 第" << this->m_Index << "轮比赛完毕 ----- " << endl;
system("pause");
}

```

- 在startSpeech比赛流程控制的函数中，调用比赛函数

```

void SpeechManager::startSpeech()
{
    //第一轮比赛
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示晋级结果

    //第二轮比赛

    //1、抽签

    //2、比赛

    //3、显示最终结果

    //4、保存分数
}

```

- 再次运行代码，测试比赛

```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
*****
***** 欢迎参加演讲比赛 *****
***** 1. 开始演讲比赛 *****
***** 2. 查看往届记录 *****
***** 3. 清空比赛记录 *****
***** 0. 退出比赛程序 *****
*****

请输入您的选择:
1
第 << 1 >> 轮比赛选手正在抽签
-----
抽签后演讲顺序如下:
10001 10002 10010 10003 10001 10012 10008 10004 10005 10007 10009 10006
-----
请按任意键继续. . .
----- 第1轮正式比赛开始: -----
第1小组比赛名次:
编号: 10001 姓名: 选手A 成绩: 87.2
编号: 10002 姓名: 选手B 成绩: 82.925
编号: 10010 姓名: 选手J 成绩: 82.9875
编号: 10011 姓名: 选手K 成绩: 79.5
编号: 10003 姓名: 选手C 成绩: 76.4625
编号: 10012 姓名: 选手L 成绩: 76.75

第2小组比赛名次:
编号: 10007 姓名: 选手G 成绩: 89.75
编号: 10008 姓名: 选手H 成绩: 86.075
编号: 10009 姓名: 选手I 成绩: 84.5375
编号: 10004 姓名: 选手D 成绩: 80.775
编号: 10006 姓名: 选手F 成绩: 80.4
编号: 10005 姓名: 选手E 成绩: 74.95

----- 第1轮比赛完毕 -----
请按任意键继续. . .
```

6.3.7 显示比赛分数

- 在speechManager.h中提供比赛的成员函数 `void showScore();`

```
//显示比赛结果
void showScore();
```

- 在speechManager.cpp中实现成员函数 `void showScore();`

```

void SpeechManager::showScore()
{
    cout << "-----第" << this->m_Index << "轮晋级选手信息如下: -----" << endl;
    vector<int>v;
    if (this->m_Index == 1)
    {
        v = v2;
    }
    else
    {
        v = vVictory;
    }

    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << "选手编号: " << *it << " 姓名: " << m_Speaker[*it].m_Name << " 得分: " <
    }
    cout << endl;

    system("pause");
    system("cls");
    this->show_Menu();
}

```

- 在startSpeech比赛流程控制的函数中，调用显示比赛分数函数

```

void SpeechManager::startSpeech()
{
    //第一轮比赛
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示晋级结果
    showScore();
    //第二轮比赛

    //1、抽签

    //2、比赛

    //3、显示最终结果

    //4、保存分数
}

```

- 运行代码，测试效果

```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
1
第 << 1 >> 轮比赛选手正在抽签
-----
抽签后演讲顺序如下:
10011 10002 10010 10003 10001 10012 10008 10004 10005 10007 10009 10006
-----
请按任意键继续. . .
----- 第1轮正式比赛开始: -----
第1小组比赛名次:
编号: 10001 姓名: 选手A 成绩: 87.2
编号: 10002 姓名: 选手B 成绩: 82.925
编号: 10010 姓名: 选手J 成绩: 82.9875
编号: 10011 姓名: 选手K 成绩: 79.5
编号: 10003 姓名: 选手C 成绩: 76.4625
编号: 10012 姓名: 选手L 成绩: 76.75

第2小组比赛名次:
编号: 10007 姓名: 选手G 成绩: 89.75
编号: 10008 姓名: 选手H 成绩: 86.075
编号: 10009 姓名: 选手I 成绩: 84.5375
编号: 10004 姓名: 选手D 成绩: 80.775
编号: 10006 姓名: 选手F 成绩: 80.4
编号: 10005 姓名: 选手E 成绩: 74.95

----- 第1轮比赛完毕 -----
请按任意键继续. . .
----- 第1轮晋级选手信息如下: -----
选手编号: 10001 姓名: 选手A 得分: 87.2
选手编号: 10002 姓名: 选手B 得分: 82.925
选手编号: 10010 姓名: 选手J 得分: 82.9875
选手编号: 10007 姓名: 选手G 得分: 89.75
选手编号: 10008 姓名: 选手H 得分: 86.075
选手编号: 10009 姓名: 选手I 得分: 84.5375

请按任意键继续. . .
```

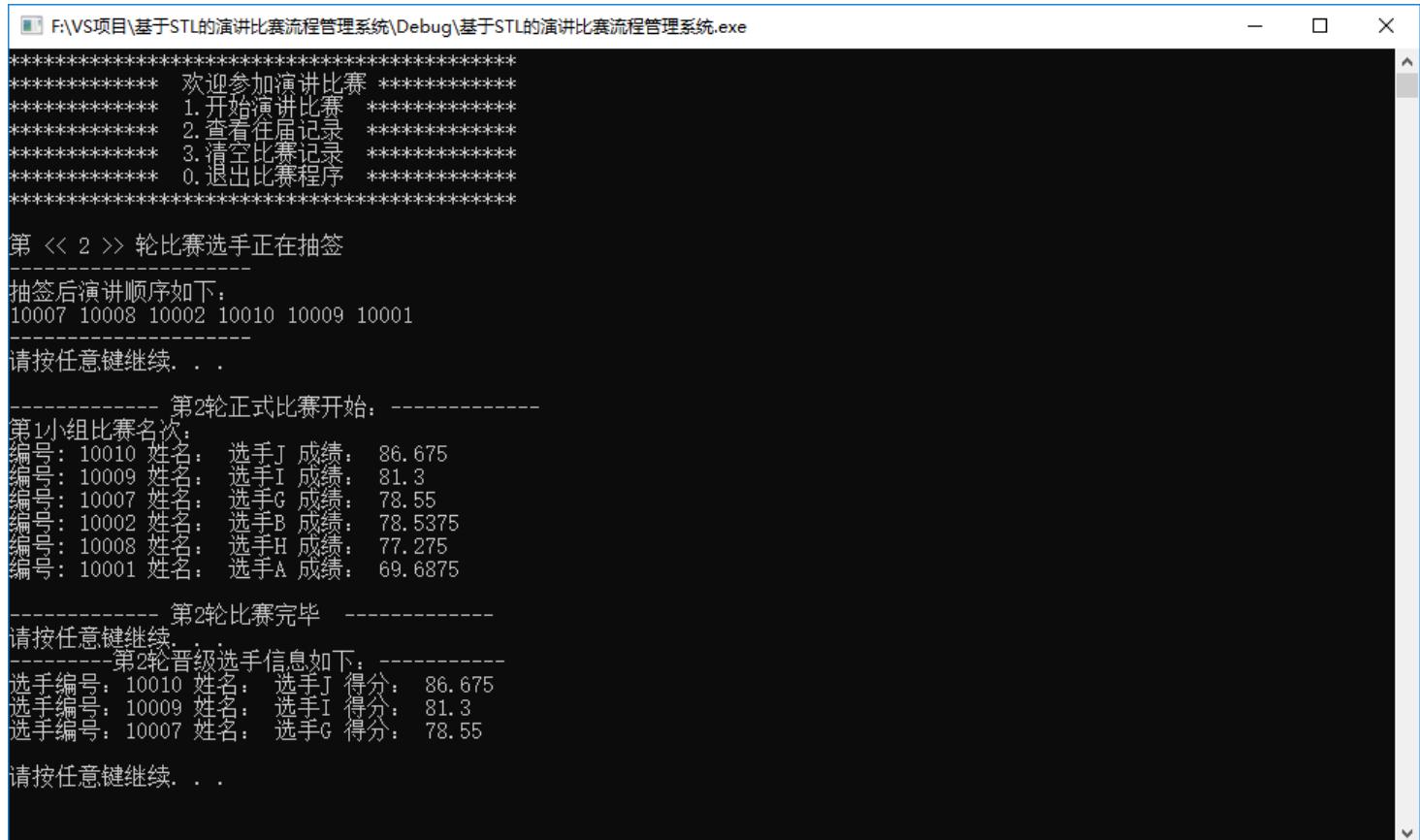
6.3.8 第二轮比赛

第二轮比赛流程同第一轮，只是比赛的轮是+1，其余流程不变

- 在startSpeech比赛流程控制的函数中，加入第二轮的流程

```
void SpeechManager::startSpeech()
{
    //第一轮比赛
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示晋级结果
    showScore();
    //第二轮比赛
    this->m_Index++;
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示最终结果
    showScore();
    //4、保存分数
}
```

测试，将整个比赛流程都跑通



```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
*****
***** 欢迎参加演讲比赛 *****
***** 1. 开始演讲比赛 *****
***** 2. 查看往届记录 *****
***** 3. 清空比赛记录 *****
***** 0. 退出比赛程序 *****
*****

第 << 2 >> 轮比赛选手正在抽签
-----
抽签后演讲顺序如下:
10007 10008 10002 10010 10009 10001
-----
请按任意键继续. . .

----- 第2轮正式比赛开始: -----
第1小组比赛名次:
编号: 10010 姓名: 选手J 成绩: 86.675
编号: 10009 姓名: 选手I 成绩: 81.3
编号: 10007 姓名: 选手G 成绩: 78.55
编号: 10002 姓名: 选手B 成绩: 78.5375
编号: 10008 姓名: 选手H 成绩: 77.275
编号: 10001 姓名: 选手A 成绩: 69.6875

----- 第2轮比赛完毕 -----
请按任意键继续. . .
----- 第2轮晋级选手信息如下. -----
选手编号: 10010 姓名: 选手J 得分: 86.675
选手编号: 10009 姓名: 选手I 得分: 81.3
选手编号: 10007 姓名: 选手G 得分: 78.55

请按任意键继续. . .
```

6.4 保存分数

功能描述：

- 将每次演讲比赛的得分记录到文件中

功能实现：

- 在speechManager.h中添加保存记录的成员函数 `void saveRecord();`

```
//保存记录  
void saveRecord();
```

- 在speechManager.cpp中实现成员函数 `void saveRecord();`

```
void SpeechManager::saveRecord()  
{  
    ofstream ofs;  
    ofs.open("speech.csv", ios::out | ios::app); // 用输出的方式打开文件 -- 写文件  
  
    //将每个人数据写入到文件中  
    for (vector<int>::iterator it = vVictory.begin(); it != vVictory.end(); it++)  
    {  
        ofs << *it << ","  
            << m_Speaker[*it].m_Score[1] << ",";  
    }  
    ofs << endl;  
  
    //关闭文件  
    ofs.close();  
  
    cout << "记录已经保存" << endl;  
}
```

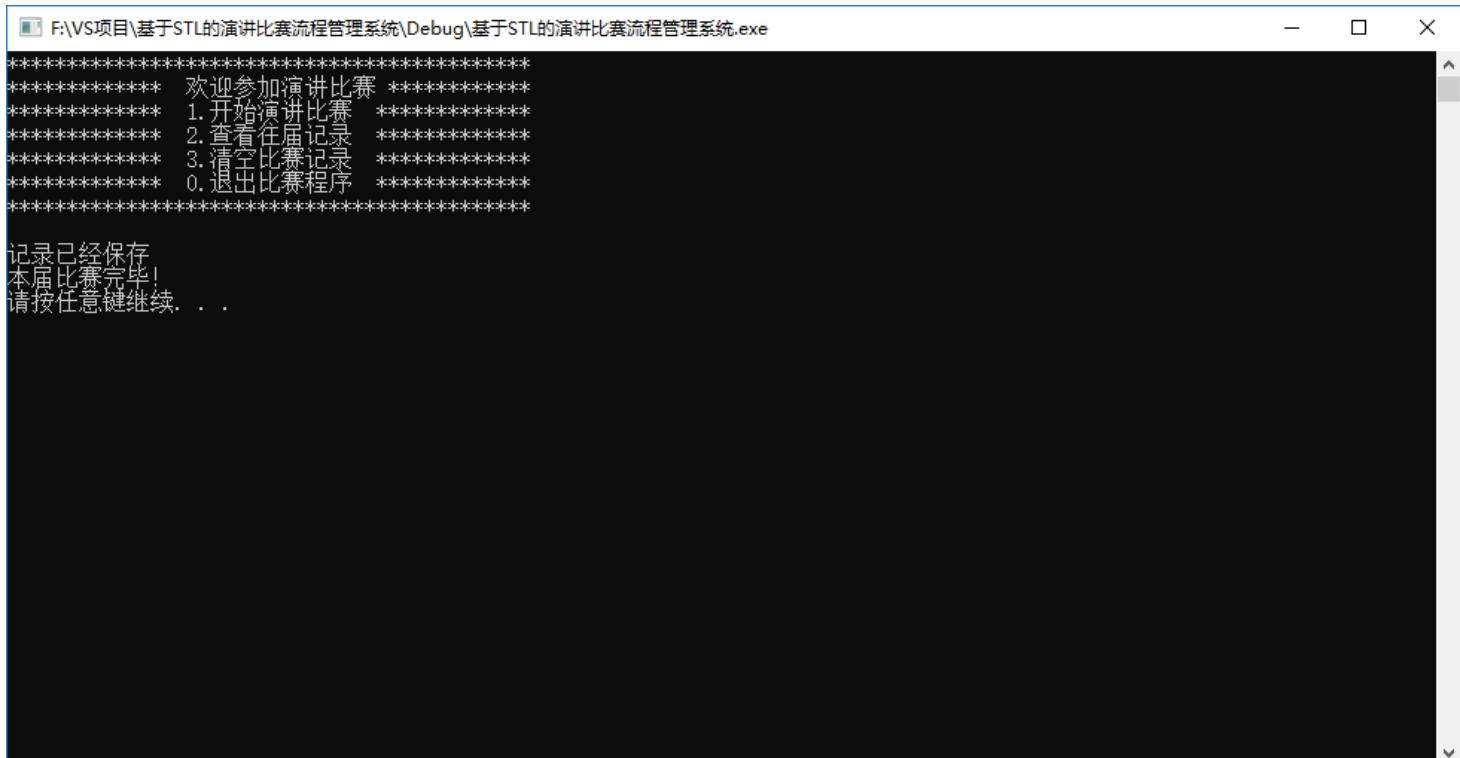
- 在startSpeech比赛流程控制的函数中，最后调用保存记录分数函数

```
void SpeechManager::startSpeech()
{
    //第一轮比赛
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示晋级结果
    showScore();
    //第二轮比赛
    this->m_Index++;
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示最终结果
    showScore();

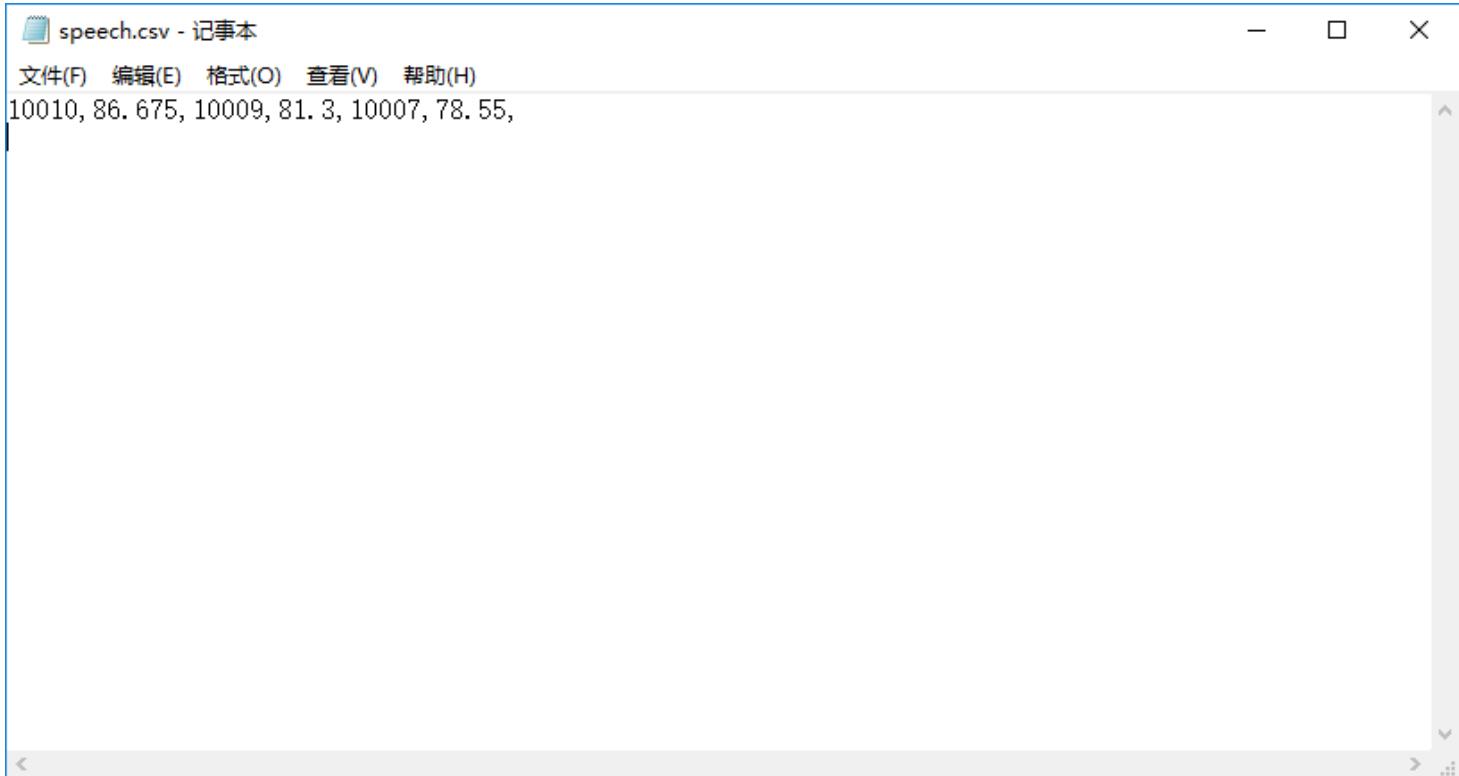
    //4、保存分数
    saveRecord();

    cout << "本届比赛完毕!" << endl;
    system("pause");
    system("cls");
}
```

- 测试，整个比赛完毕后记录保存情况



利用记事本打开文件 speech.csv，里面保存了前三名选手的编号以及得分



至此，整个演讲比赛功能制作完毕！

7、查看记录

7.1 读取记录分数

- 在speechManager.h中添加保存记录的成员函数 `void loadRecord();`
- 添加判断文件是否为空的标志 `bool fileIsEmpty;`
- 添加往届记录的容器 `map<int, vector<string>> m_Record;`

其中`m_Record` 中的key代表第几届， value记录具体的信息

```
//读取记录
void loadRecord();

//文件为空的标志
bool fileIsEmpty;

//往届记录
map<int, vector<string>> m_Record;
```

- 在speechManager.cpp中实现成员函数 `void loadRecord();`

```
void SpeechManager::loadRecord()
{
    ifstream ifs("speech.csv", ios::in); //输入流对象 读取文件

    if (!ifs.is_open())
    {
        this->fileIsEmpty = true;
        cout << "文件不存在! " << endl;
        ifs.close();
        return;
    }

    char ch;
    ifs >> ch;
    if (ifs.eof())
    {
        cout << "文件为空!" << endl;
        this->fileIsEmpty = true;
        ifs.close();
        return;
    }

    //文件不为空
    this->fileIsEmpty = false;

    ifs.putback(ch); //读取的单个字符放回去

    string data;
    int index = 0;
    while (ifs >> data)
    {
        //cout << data << endl;
        vector<string>v;

        int pos = -1;
        int start = 0;

        while (true)
        {
            pos = data.find(",", start); //从0开始查找 ','
            if (pos == -1)
            {
                break; //找不到break返回
            }
            string tmp = data.substr(start, pos - start); //找到了,进行分割 参数1 起始
            v.push_back(tmp);
            start = pos + 1;
        }

        this->m_Record.insert(make_pair(index, v));
        index++;
    }
}
```

```
    ifs.close();
}
```

- 在SpeechManager构造函数中调用获取往届记录函数

```
SpeechManager::SpeechManager()
{
    //初始化属性
    this->initSpeech();

    //创建选手
    this->createSpeaker();

    //获取往届记录
    this->loadRecord();
}
```

7.2 查看记录功能

- 在speechManager.h中添加保存记录的成员函数 void showRecord();

```
//显示往届得分
void showRecord();
```

- 在speechManager.cpp中实现成员函数 void showRecord();

```
void SpeechManager::showRecord()
{
    for (int i = 0; i < this->m_Record.size(); i++)
    {
        cout << "第" << i + 1 << "届" <<
        "冠军编号：" << this->m_Record[i][0] << "得分：" << this->m_Record[i][1]
        "亚军编号：" << this->m_Record[i][2] << "得分：" << this->m_Record[i][3]
        "季军编号：" << this->m_Record[i][4] << "得分：" << this->m_Record[i][5]
    }
    system("pause");
    system("cls");
}
```

7.3 测试功能

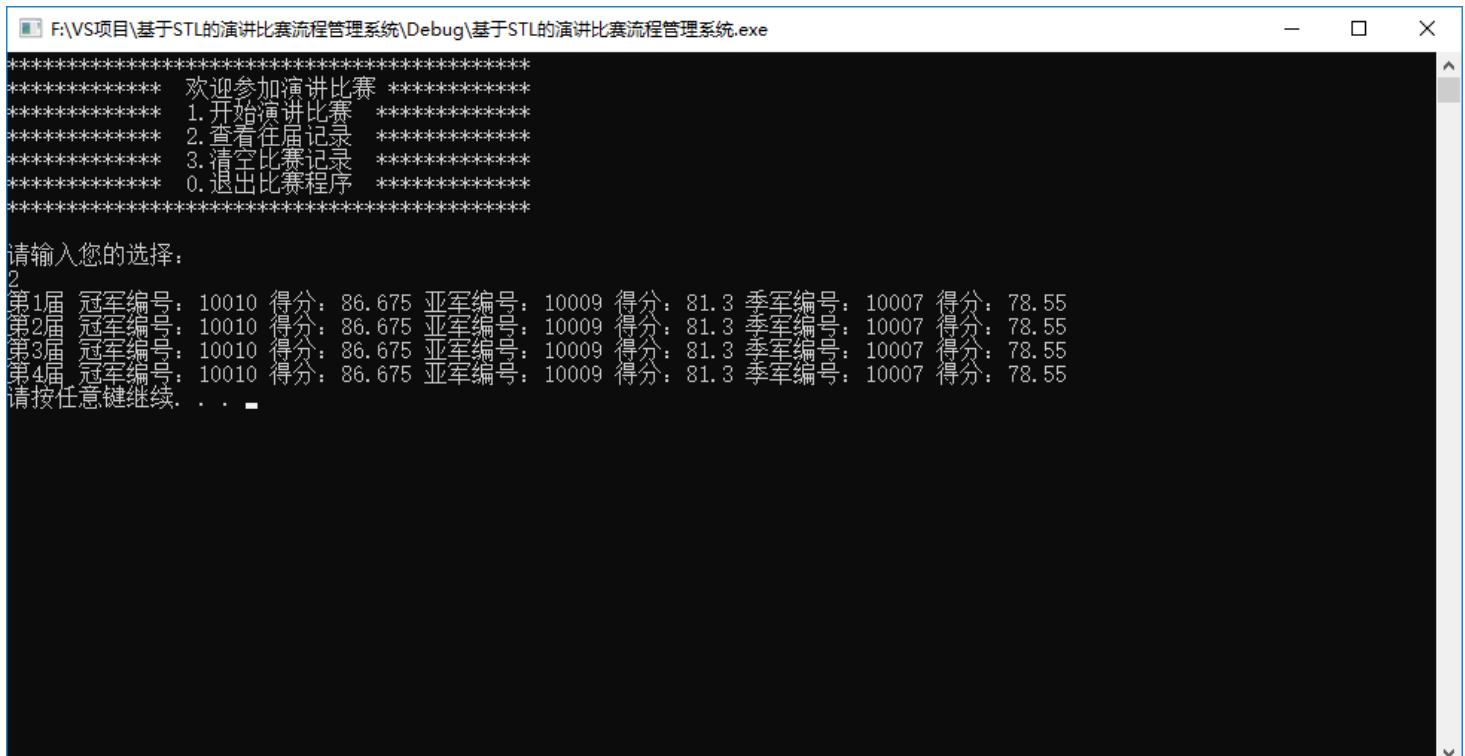
在main函数分支 2 选项中，调用查看记录的接口

```
while (true)
{
    sm.show_Menu();

    cout << "请输入您的选择: " << endl;
    cin >> choice; // 接受用户的选项

    switch (choice)
    {
        case 1: //开始比赛
            sm.startSpeech();
            break;
        case 2: //查看记录
            sm.showRecord();
            break;
        case 3: //清空记录
            break;
        case 0: //退出系统
            sm.exitSystem();
            break;
        default:
            system("cls"); //清屏
            break;
    }
}
```

显示效果如图：（本次测试添加了4条记录）



7.4 bug解决

目前程序中有几处bug未解决：

1. 查看往届记录，若文件不存在或为空，并未提示

解决方式：在showRecord函数中，开始判断文件状态并加以判断

```
void SpeechManager::showRecord()
{
    if (this->fileIsEmpty)
    {
        cout << "文件不存在，或记录为空！" << endl;
    }
    else
    {
        for (int i = 0; i < this->m_Record.size(); i++)
        {
            cout << "第" << i + 1 << "届" <<
                "冠军编号：" << this->m_Record[i][0] << "得分：" << this->m_Record[i][1]
                "亚军编号：" << this->m_Record[i][2] << "得分：" << this->m_Record[i][3]
                "季军编号：" << this->m_Record[i][4] << "得分：" << this->m_Record[i][5]
        }
    }
    system("pause");
    system("cls");
}
```

2. 若记录为空或不存在，比完赛后依然提示记录为空

解决方式：saveRecord中更新文件为空的标志

```
void SpeechManager::saveRecord()
{
    ofstream ofs;
    ofs.open("speech.csv", ios::out | ios::app); // 用输出的方式打开文件 -- 写文件

    //将每个人数据写入到文件中
    for (vector<int>::iterator it = vVictory.begin(); it != vVictory.end(); it++)
    {
        ofs << *it << ","
            << m_Speaker[*it].m_Score[1] << ",";
    }
    ofs << endl;

    //关闭文件
    ofs.close();

    cout << "记录已经保存" << endl;
}

//有记录了，文件不为空
this->fileIsEmpty = false;
```

3. 比赛后查不到本届比赛的记录，没有实时更新

解决方式：比赛完毕后，所有数据重置

```
void SpeechManager::startSpeech()
{
    //第一轮比赛
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示晋级结果
    showScore();
    //第二轮比赛
    this->m_Index++;
    //1、抽签
    speechDraw();
    //2、比赛
    speechContest();
    //3、显示最终结果
    showScore();

    //4、保存分数
    saveRecord();
    //重置比赛
    //初始化属性
    this->initSpeech();

    //创建选手
    this->createSpeaker();

    //获取往届记录
    this->loadRecord();

    cout << "本届比赛完毕!" << endl;
    system("pause");
    system("cls");
}
```

4. 在初始化时，没有初始化记录容器

解决方式：initSpeech中添加 初始化记录容器

```
void SpeechManager::initSpeech()
{
    //容器保证为空
    this->v1.clear();
    this->v2.clear();
    this->vVictory.clear();
    this->m_Speaker.clear();
    //初始化比赛轮数
    this->m_Index = 1;
    //初始化记录容器
    this->m_Record.clear();
}
```

5. 每次记录都是一样的

解决方式：在main函数一开始 添加随机数种子

```
 srand((unsigned int)time(NULL));
```

所有bug解决后 测试：



```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
*****
***** 欢迎参加演讲比赛 *****
***** 1. 开始演讲比赛 *****
***** 2. 查看往届记录 *****
***** 3. 清空比赛记录 *****
***** 0. 退出比赛程序 *****
*****

请输入您的选择:
2
第1届 冠军编号: 10002 得分: 88.15 亚军编号: 10006 得分: 83.225 季军编号: 10011 得分: 82.075
第2届 冠军编号: 10003 得分: 88.9875 亚军编号: 10011 得分: 86.4 季军编号: 10009 得分: 85.6375
第3届 冠军编号: 10009 得分: 82.775 亚军编号: 10011 得分: 81.5375 季军编号: 10003 得分: 78.4125
第4届 冠军编号: 10004 得分: 84.875 亚军编号: 10011 得分: 83.6875 季军编号: 10005 得分: 82.85
第5届 冠军编号: 10006 得分: 84.75 亚军编号: 10005 得分: 83.95 季军编号: 10004 得分: 79.525
请按任意键继续... . .
```

8、清空记录

8.1 清空记录功能实现

- 在speechManager.h中添加保存记录的成员函数 void clearRecord();

```
//清空记录
```

```
void clearRecord();
```

- 在speechManager.cpp中实现成员函数 void clearRecord();

```
void SpeechManager::clearRecord()
{
    cout << "确认清空? " << endl;
    cout << "1、确认" << endl;
    cout << "2、返回" << endl;

    int select = 0;
    cin >> select;

    if (select == 1)
    {
        //打开模式 ios::trunc 如果存在删除文件并重新创建
        ofstream ofs("speech.csv", ios::trunc);
        ofs.close();

        //初始化属性
        this->initSpeech();

        //创建选手
        this->createSpeaker();

        //获取往届记录
        this->loadRecord();

        cout << "清空成功! " << endl;
    }

    system("pause");
    system("cls");
}
```

8.2 测试清空

在main函数分支 3 选项中，调用清空比赛记录的接口

```
while (true)
{
    sm.show_Menu();

    cout << "请输入您的选择: " << endl;
    cin >> choice; // 接受用户的选项

    switch (choice)
    {
        case 1: //开始比赛
            sm.startSpeech();
            break;
        case 2: //查看记录
            sm.showRecord();
            break;
        case 3: //清空记录
            sm.clearRecord();
            break;
        case 0: //退出系统
            sm.exitSystem();
            break;
        default:
            system("cls"); //清屏
            break;
    }
}
```

运行程序，测试清空记录：

```
F:\VS项目\基于STL的演讲比赛流程管理系统\Debug\基于STL的演讲比赛流程管理系统.exe
*****
***** 欢迎参加演讲比赛 *****
***** 1. 开始演讲比赛 *****
***** 2. 查看往届记录 *****
***** 3. 清空比赛记录 *****
***** 0. 退出比赛程序 *****
*****
请输入您的选择:
3
确认清空?
1、确认
2、返回
1
清空成功!
请按任意键继续. . .
```

speech.csv中记录也为空



- 至此本案例结束! ^_^

机房预约系统

1、机房预约系统需求

1.1 系统简介

- 学校现有几个规格不同的机房，由于使用时经常出现“撞车”现象，现开发一套机房预约系统，解决这一问题。



1.2 身份简介

分别有三种身份使用该程序

- **学生代表**: 申请使用机房
- **教师**: 审核学生的预约申请
- **管理员**: 给学生、教师创建账号

1.3 机房简介

机房总共有3间

- 1号机房 --- 最大容量20人
- 2号机房 --- 最多容量50人
- 3号机房 --- 最多容量100人

1.4 申请简介

- 申请的订单每周由管理员负责清空。
- 学生可以预约未来一周内的机房使用，预约的日期为周一至周五，预约时需要选择预约时段（上午、下午）
- 教师来审核预约，依据实际情况审核预约通过或者不通过

1.5 系统具体需求

- 首先进入登录界面，可选登录身份有：
 - 学生代表
 - 老师
 - 管理员
 - 退出
- 每个身份都需要进行验证后，进入子菜单
 - 学生需要输入：学号、姓名、登录密码
 - 老师需要输入：职工号、姓名、登录密码
 - 管理员需要输入：管理员姓名、登录密码
- 学生具体功能
 - 申请预约 --- 预约机房
 - 查看自身的预约 --- 查看自己的预约状态
 - 查看所有预约 --- 查看全部预约信息以及预约状态
 - 取消预约 --- 取消自身的预约，预约成功或审核中的预约均可取消
 - 注销登录 --- 退出登录
- 教师具体功能
 - 查看所有预约 --- 查看全部预约信息以及预约状态
 - 审核预约 --- 对学生的预约进行审核
 - 注销登录 --- 退出登录
- 管理员具体功能
 - 添加账号 --- 添加学生或教师的账号，需要检测学生编号或教师职工号是否重复
 - 查看账号 --- 可以选择查看学生或教师的全部信息
 - 查看机房 --- 查看所有机房的信息
 - 清空预约 --- 清空所有预约记录
 - 注销登录 --- 退出登录



2、创建项目

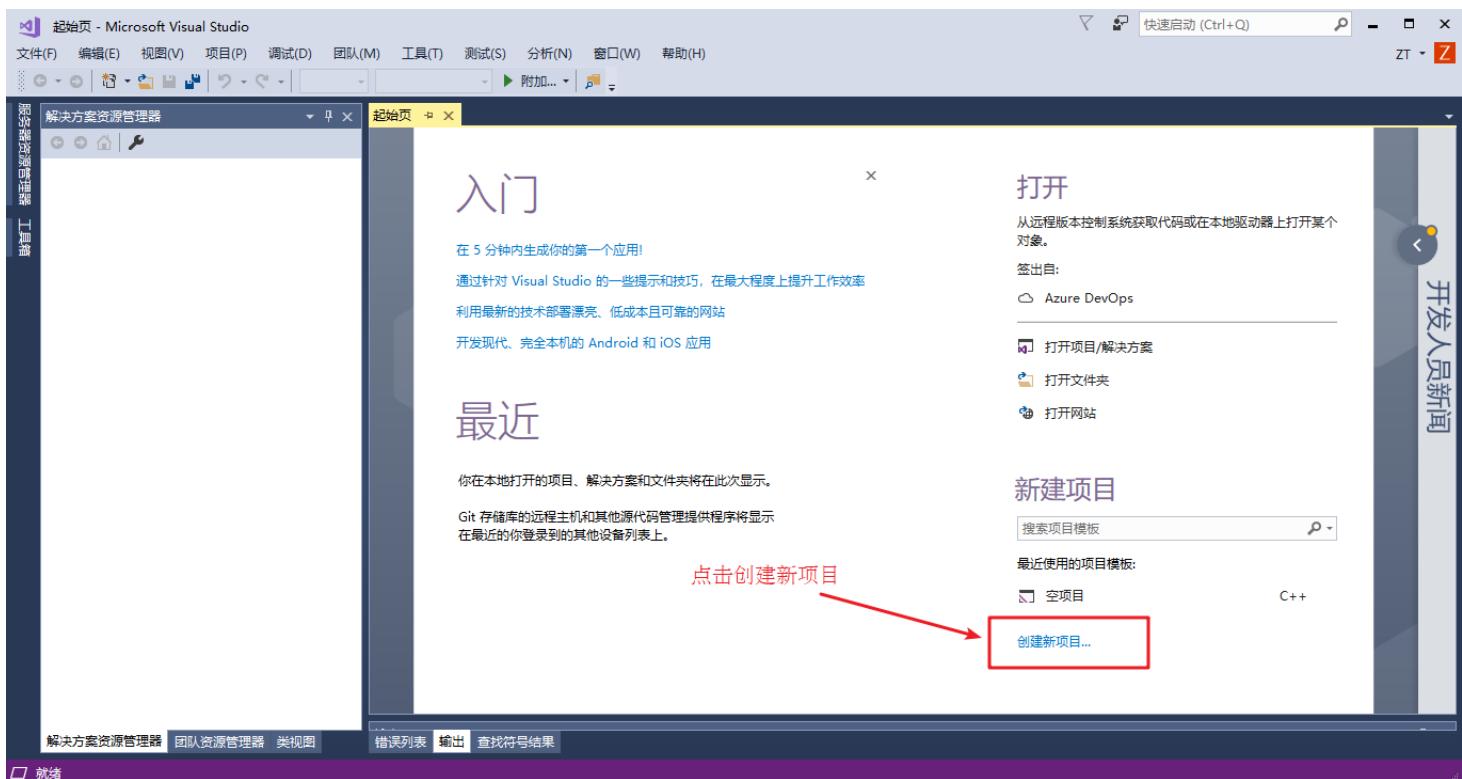
创建项目步骤如下：

- 创建新项目
- 添加文件

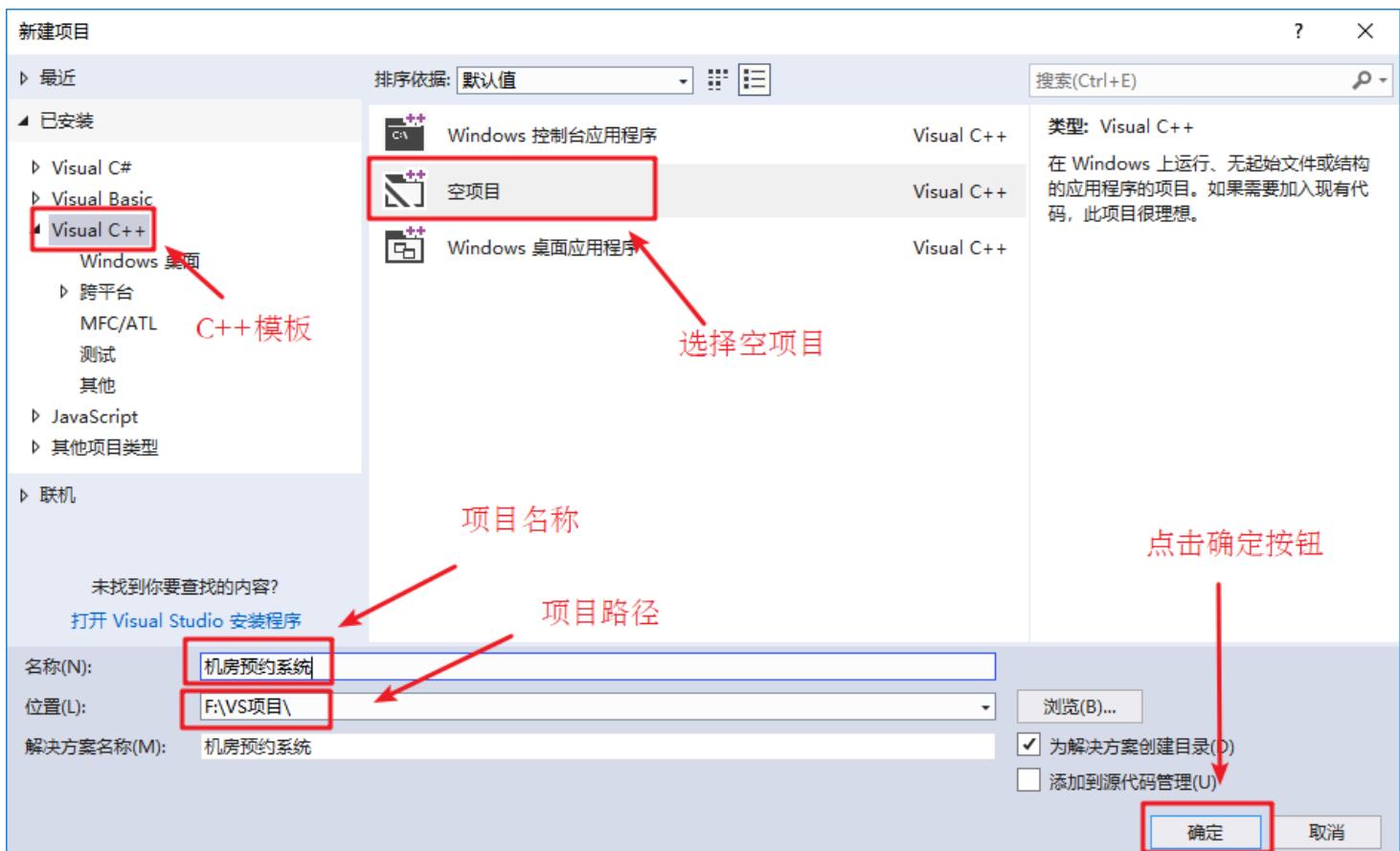
2.1 创建项目

- 打开vs2017后，点击创建新项目，创建新的C++项目

如图：

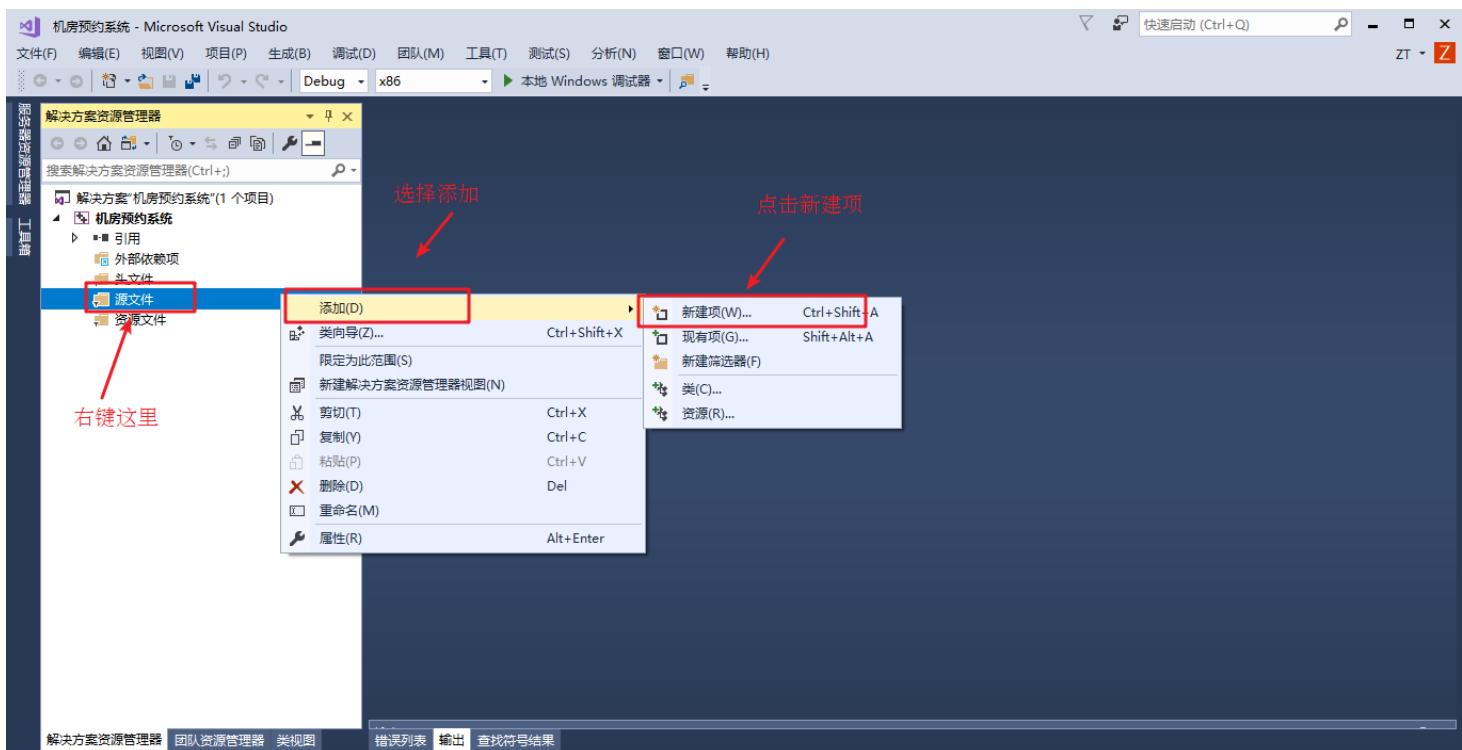


- 填写项目名称以及选取项目路径，点击确定生成项目

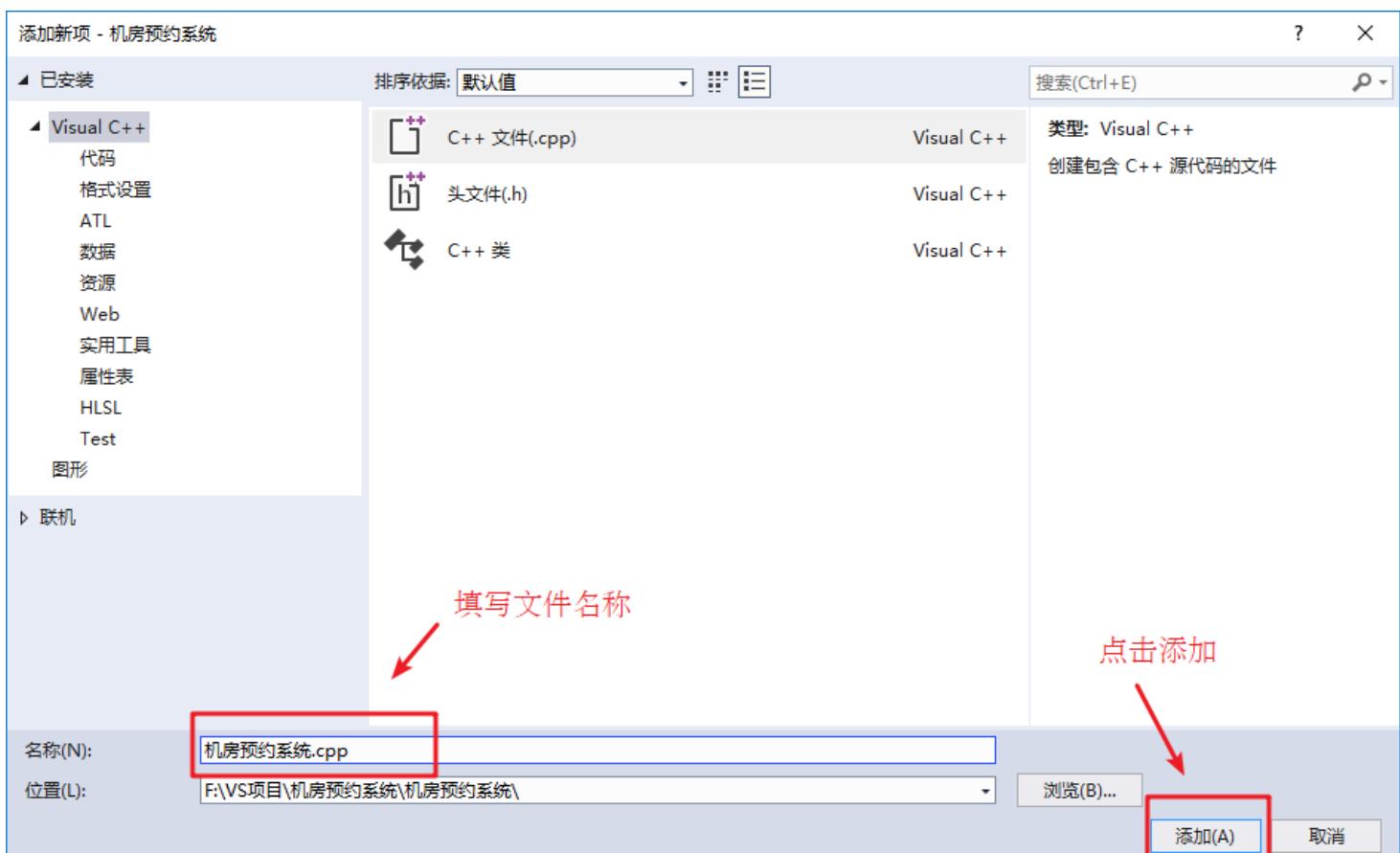


2.2 添加文件

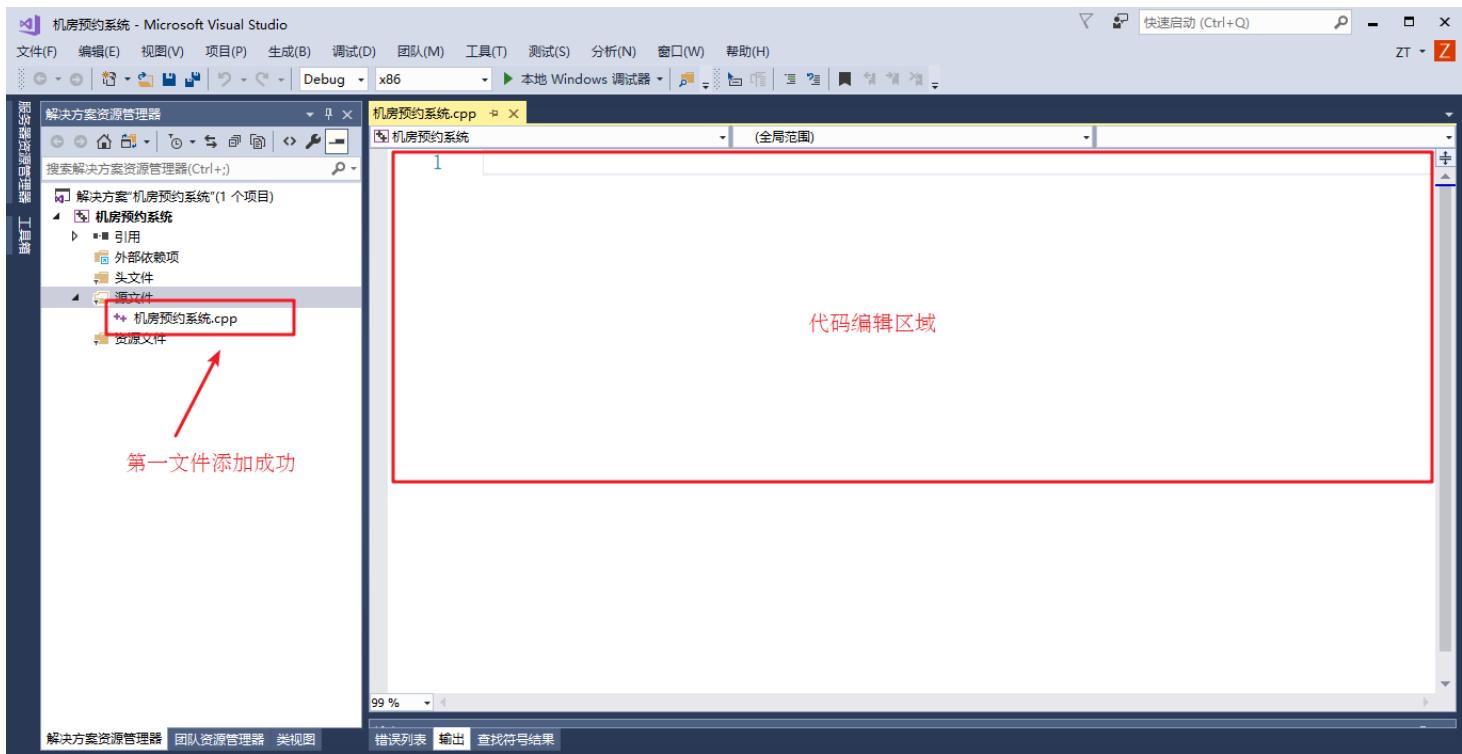
- 右键源文件，进行添加文件操作



- 填写文件名称，点击添加



- 生成文件成功，效果如下图



3、创建主菜单

功能描述：

- 设计主菜单，与用户进行交互

3.1 菜单实现

- 在主函数main中添加菜单提示，代码如下：

```
int main() {  
  
    cout << "===== 欢迎来到传智播客机房预约系统 ====="  
    << endl;  
    cout << endl << "请输入您的身份" << endl;  
    cout << "\t\t ----- \n";  
    cout << "\t\t| \n";  
    cout << "\t\t| 1. 学生代表 | \n";  
    cout << "\t\t| \n";  
    cout << "\t\t| 2. 老师 | \n";  
    cout << "\t\t| \n";  
    cout << "\t\t| 3. 管理员 | \n";  
    cout << "\t\t| \n";  
    cout << "\t\t| 0. 退出 | \n";  
    cout << "\t\t| \n";  
    cout << "\t\t ----- \n";  
    cout << "输入您的选择: ";  
  
    system("pause");  
  
    return 0;  
}
```

运行效果如图：



3.2 搭建接口

- 接受用户的选择，搭建接口
- 在main中添加代码

```

int main() {

    int select = 0;

    while (true)
    {

        cout << "===== 欢迎来到传智播客机房预约系统 =====";
        cout << endl << "请输入您的身份" << endl;
        cout << "\t\t-----\n";
        cout << "\t\t|\n";
        cout << "\t\t| 1. 学生代表\n";
        cout << "\t\t| |\n";
        cout << "\t\t| 2. 老师\n";
        cout << "\t\t| |\n";
        cout << "\t\t| 3. 管理员\n";
        cout << "\t\t| |\n";
        cout << "\t\t| 0. 退出\n";
        cout << "\t\t| |\n";
        cout << "\t\t-----\n";
        cout << "输入您的选择: ";

        cin >> select; //接受用户选择

        switch (select)
        {
        case 1: //学生身份
            break;
        case 2: //老师身份
            break;
        case 3: //管理员身份
            break;
        case 0: //退出系统
            break;
        default:
            cout << "输入有误, 请重新选择!" << endl;
            system("pause");
            system("cls");
            break;
        }

    }
    system("pause");
    return 0;
}

```

测试，输入0、1、2、3会重新回到界面，输入其他提示输入有误，清屏后重新选择

效果如图：



至此，界面搭建完毕

4、退出功能实现

4.1 退出功能实现

在main函数分支0选项中，添加退出程序的代码：

```
cout << "欢迎下一次使用" << endl;
system("pause");
return 0;
```

```
switch (select)
{
case 1: //学生身份
    break;
case 2: //老师身份
    break;
case 3: //管理员身份
    break;
case 0: //退出系统
    cout << "欢迎下一次使用" << endl;
    system("pause");
    return 0;
    break;
default:
    cout << "输入有误, 请重新选择!" << endl;
    system("pause");
    system("cls");
    break;
}
```

4.2 测试退出功能

运行程序，效果如图：



至此，退出程序功能实现

5、 创建身份类

5.1 身份的基类

- 在整个系统中，有三种身份，分别为：学生代表、老师以及管理员
- 三种身份有其共性也有其特性，因此我们可以将三种身份抽象出一个身份基类**identity**
- 在头文件下创建Identity.h文件

Identity.h中添加如下代码：

```
#pragma once
#include<iostream>
using namespace std;

//身份抽象类
class Identity
{
public:
    //操作菜单
    virtual void operMenu() = 0;

    string m_Name; //用户名
    string m_Pwd; //密码
};
```

效果如图：

The screenshot shows a code editor with two tabs: 'identity.h' and '机房预约系统.cpp'. The 'identity.h' tab is active, displaying the following C++ code:

```
1 #pragma once
2 #include<iostream>
3 using namespace std;
4
5 //身份抽象类
6 class Identity
7 {
8 public:
9
10    //操作菜单
11    virtual void operMenu() = 0;
12
13    string m_Name; //用户名
14    string m_Pwd; //密码
15};
```

A red rectangular box highlights the entire class definition from line 6 to line 15.

5.2 学生类

5.2.1 功能分析

- 学生类主要功能是可以通过类中成员函数，实现预约实验室操作
- 学生类中主要功能有：
 - 显示学生操作的菜单界面
 - 申请预约
 - 查看自身预约
 - 查看所有预约
 - 取消预约

5.2.2 类的创建

- 在头文件以及源文件下创建 student.h 和 student.cpp 文件

student.h 中添加如下代码：

```
#pragma once
#include<iostream>
using namespace std;
#include "identity.h"

//学生类
class Student :public Identity
{
public:
    //默认构造
    Student();

    //有参构造(学号、姓名、密码)
    Student(int id, string name, string pwd);

    //菜单界面
    virtual void operMenu();

    //申请预约
    void applyOrder();

    //查看我的预约
    void showMyOrder();

    //查看所有预约
    void showAllOrder();

    //取消预约
    void cancelOrder();

    //学生学号
    int m_Id;

};

};


```

student.cpp中添加如下代码：

```
#include "student.h"

//默认构造
Student::Student()
{
}

//有参构造(学号、姓名、密码)
Student::Student(int id, string name, string pwd)
{
}

//菜单界面
void Student::operMenu()
{
}

//申请预约
void Student::applyOrder()
{
}

//查看我的预约
void Student::showMyOrder()
{
}

//查看所有预约
void Student::showAllOrder()
{
}

//取消预约
void Student::cancelOrder()
{
}
```

5.3 老师类

5.3.1 功能分析

- 教师类主要功能是查看学生的预约，并进行审核
- 教师类中主要功能有：
 - 显示教师操作的菜单界面
 - 查看所有预约
 - 审核预约

5.3.2 类的创建

- 在头文件以及源文件下创建 teacher.h 和 teacher.cpp 文件

teacher.h 中添加如下代码：

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
#include "identity.h"

class Teacher :public Identity
{
public:

    //默认构造
    Teacher();

    //有参构造 (职工编号, 姓名, 密码)
    Teacher(int empId, string name, string pwd);

    //菜单界面
    virtual void operMenu();

    //查看所有预约
    void showAllOrder();

    //审核预约
    void validOrder();

    int m_EmpId; //教师编号
};

};
```

- teacher.cpp 中添加如下代码：

```
#include"teacher.h"

//默认构造
Teacher::Teacher()
{
}

//有参构造 (职工编号, 姓名, 密码)
Teacher::Teacher(int empId, string name, string pwd)
{
}

//菜单界面
void Teacher::operMenu()
{
}

//查看所有预约
void Teacher::showAllOrder()
{
}

//审核预约
void Teacher::validOrder()
{
}
```

5.4 管理员类

5.4.1 功能分析

- 管理员类主要功能是对学生和老师账户进行管理，查看机房信息以及清空预约记录
- 管理员类中主要功能有：
 - 显示管理员操作的菜单界面
 - 添加账号
 - 查看账号
 - 查看机房信息
 - 清空预约记录

5.4.2 类的创建

- 在头文件以及源文件下创建 manager.h 和 manager.cpp文件

manager.h中添加如下代码：

```
#pragma once
#include<iostream>
using namespace std;
#include "identity.h"

class Manager :public Identity
{
public:

    //默认构造
    Manager();

    //有参构造 管理员姓名，密码
    Manager(string name, string pwd);

    //选择菜单
    virtual void operMenu();

    //添加账号
    void addPerson();

    //查看账号
    void showPerson();

    //查看机房信息
    void showComputer();

    //清空预约记录
    void cleanFile();

};

};
```

- manager.cpp中添加如下代码:

```
#include "manager.h"

//默认构造
Manager::Manager()
{
}

//有参构造
Manager::Manager(string name, string pwd)
{
}

//选择菜单
void Manager::operMenu()
{
}

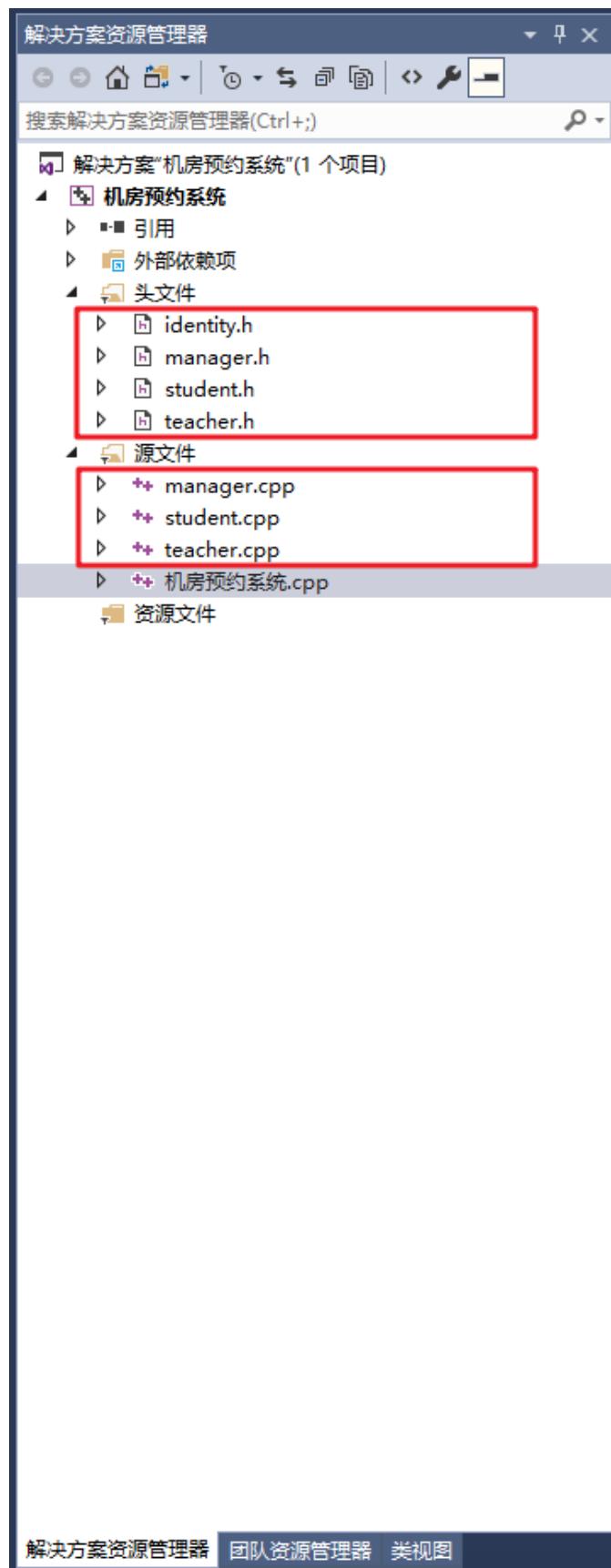
//添加账号
void Manager::addPerson()
{
}

//查看账号
void Manager::showPerson()
{
}

//查看机房信息
void Manager::showComputer()
{
}

//清空预约记录
void Manager::cleanFile()
{
}
```

至此，所有身份类创建完毕，效果如图：



6、登录模块

6.1 全局文件添加

功能描述：

- 不同的身份可能会用到不同的文件操作，我们可以将所有的文件名定义到一个全局的文件中
- 在头文件中添加 **globalFile.h** 文件
- 并添加如下代码：

```
#pragma once

//管理员文件
#define ADMIN_FILE      "admin.txt"
//学生文件
#define STUDENT_FILE    "student.txt"
//教师文件
#define TEACHER_FILE    "teacher.txt"
//机房信息文件
#define COMPUTER_FILE   "computerRoom.txt"
//订单文件
#define ORDER_FILE       "order.txt"
```

并且在同级目录下，创建这几个文件

名称	修改日期	类型	大小
Debug	2019/1/27 11:14	文件夹	
globalFile.h	2019/1/27 15:51	C/C++ Header	1 KB
identity.h	2019/1/27 15:14	C/C++ Header	1 KB
manager.h	2019/1/27 15:30	C/C++ Header	1 KB
student.h	2019/1/27 15:20	C/C++ Header	1 KB
teacher.h	2019/1/27 15:23	C/C++ Header	1 KB
* manager.cpp	2019/1/27 15:31	C++ Source	1 KB
* student.cpp	2019/1/27 15:21	C++ Source	1 KB
* teacher.cpp	2019/1/27 15:25	C++ Source	1 KB
* 机房预约系统.cpp	2019/1/27 15:52	C++ Source	4 KB
机房预约系统.vcxproj.user	2019/1/27 10:19	Per-User Project...	1 KB
机房预约系统.vcxproj	2019/1/27 10:40	VC++ Project	6 KB
机房预约系统.vcxproj.filters	2019/1/27 10:40	VC++ Project Fil...	1 KB
admin.txt	2019/1/27 15:53	文本文档	0 KB
computerRoom.txt	2019/1/27 15:53	文本文档	0 KB
order.txt	2019/1/26 13:54	文本文档	0 KB
student.txt	2019/1/27 15:53	文本文档	0 KB
teacher.txt	2019/1/27 15:53	文本文档	0 KB

6.2 登录函数封装

功能描述：

- 根据用户的选择，进入不同的身份登录

在预约系统的.cpp文件中添加全局函数 `void LoginIn(string fileName, int type)`

参数：

- `fileName` --- 操作的文件名
- `type` --- 登录的身份 (1代表学生、2代表老师、3代表管理员)

`LoginIn`中添加如下代码：

```
#include "globalFile.h"
#include "identity.h"
#include <fstream>
#include <string>

//登录功能
void LoginIn(string fileName, int type)
{
    Identity * person = NULL;

    ifstream ifs;
    ifs.open(fileName, ios::in);

    //文件不存在情况
    if (!ifs.is_open())
    {
        cout << "文件不存在" << endl;
        ifs.close();
        return;
    }

    int id = 0;
    string name;
    string pwd;

    if (type == 1) //学生登录
    {
        cout << "请输入你的学号" << endl;
        cin >> id;
    }
    else if (type == 2) //教师登录
    {
        cout << "请输入你的职工号" << endl;
        cin >> id;
    }

    cout << "请输入用户名：" << endl;
    cin >> name;

    cout << "请输入密码：" << endl;
    cin >> pwd;

    if (type == 1)
    {
        //学生登录验证
    }
    else if (type == 2)
    {
        //教师登录验证
    }
    else if(type == 3)
    {
```

```
//管理员登录验证
}

cout << "验证登录失败!" << endl;

system("pause");
system("cls");
return;
}
```

- 在main函数的不同分支中，填入不同的登录接口

```
switch (select)
{
case 1: //学生身份
    LoginIn(STUDENT_FILE, 1);
    break;
case 2: //老师身份
    LoginIn(TEACHER_FILE, 2);
    break;
case 3: //管理员身份
    LoginIn(ADMIN_FILE, 3);
    break;
case 0: //退出系统
    cout << "欢迎下一次使用" << endl;
    system("pause");
    return 0;
    break;
default:
    cout << "输入有误，请重新选择！" << endl;
    system("pause");
    system("cls");
    break;
}
```

6.3 学生登录实现

在student.txt文件中添加两条学生信息，用于测试

添加信息：

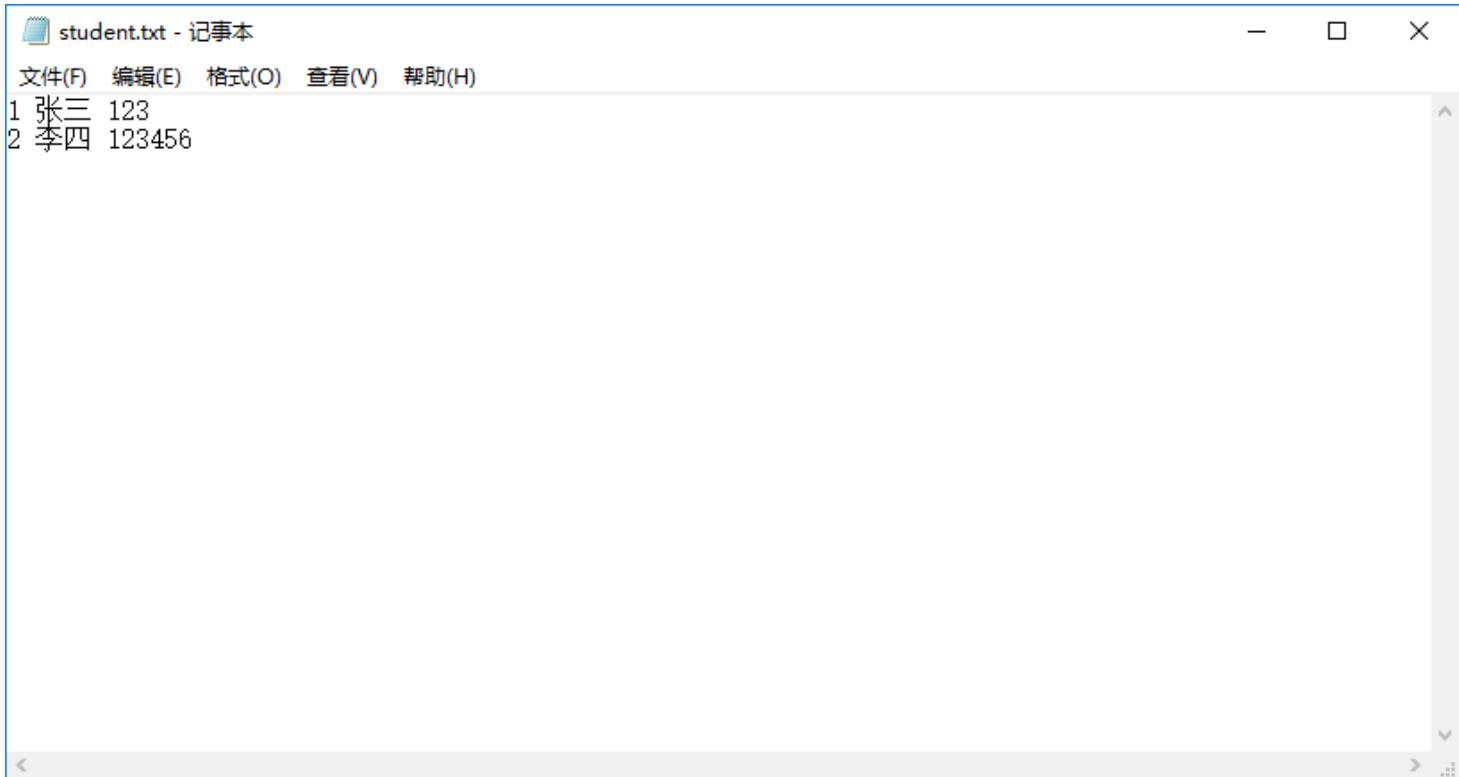
```
1 张三 123
2 李四 123456
```

其中：

- 第一列代表学号

- 第二列 代表 学生姓名
- 第三列 代表 密码

效果图：



在Login函数的学生分支中加入如下代码，验证学生身份

```
//学生登录验证
int fId;
string fName;
string fPwd;
while (ifs >> fId && ifs >> fName && ifs >> fPwd)
{
    if (id == fId && name == fName && pwd == fPwd)
    {
        cout << "学生验证登录成功!" << endl;
        system("pause");
        system("cls");
        person = new Student(id, name, pwd);

        return;
    }
}
```

添加代码效果图

```
if (type == 1)
{
    //学生登录验证
    int fId;
    string fName;
    string fPwd;
    while (ifs >> fId && ifs >> fName && ifs >> fPwd)
    {
        if (id == fId && name == fName && pwd == fPwd)
        {
            cout << "学生验证登录成功!" << endl;
            system("pause");
            system("cls");
            person = new Student(id, name, pwd);

            return;
        }
    }
}

else if (type == 2)
{
    //教师登录验证
}
else if(type == 3)
{
    //管理员登录验证
}
```

测试：

F:\VS项目\机房预约系统\Debug\机房预约系统.exe
===== 欢迎来到传智播客机房预约系统 =====
请输入您的身份
1. 学生代表
2. 老师
3. 管理员
0. 退出
输入您的选择: 1
请输入你的学号
1
请输入用户名:
张三
请输入密码:
123
学生验证登录成功!
请按任意键继续... .

6.4 教师登录实现

在teacher.txt文件中添加一条老师信息，用于测试

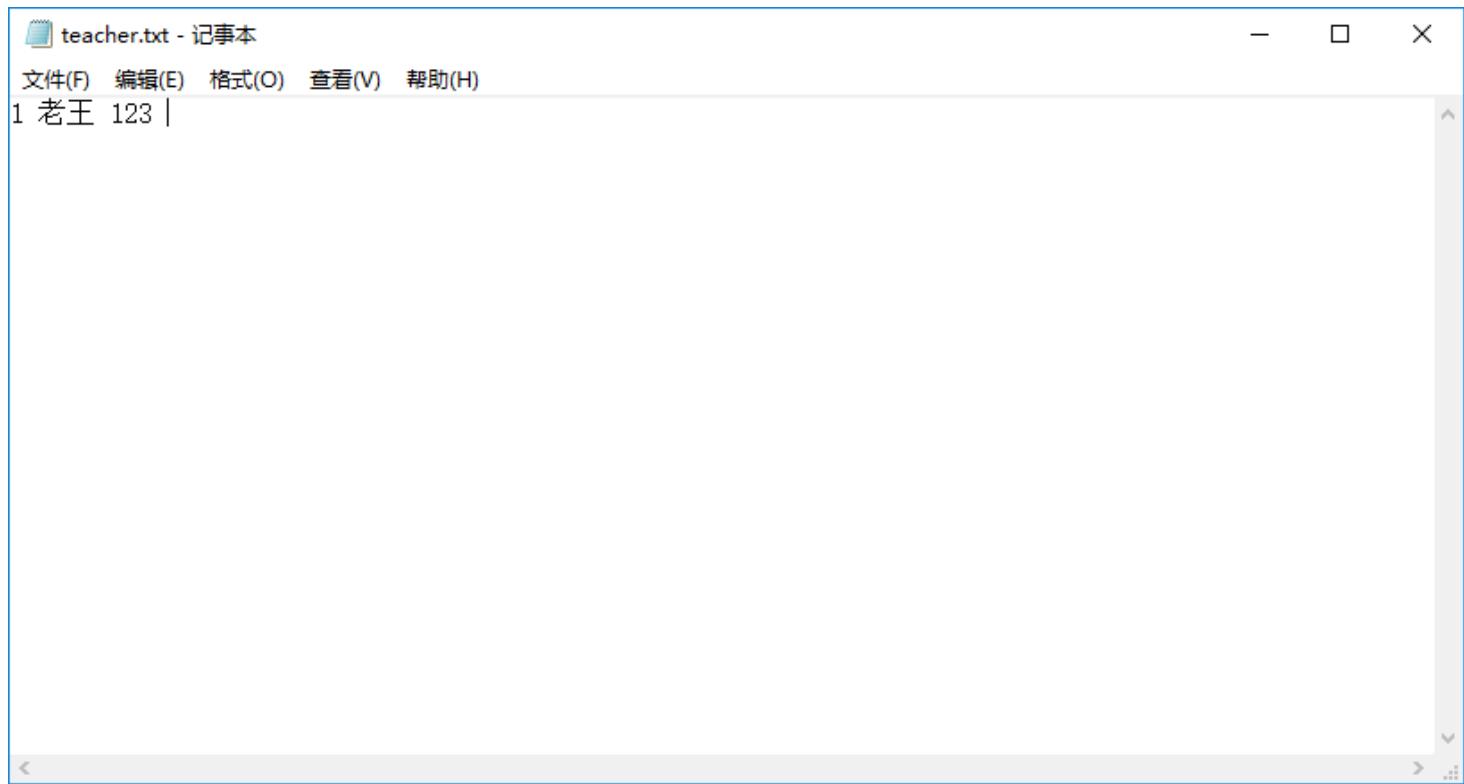
添加信息:

1 老王 123

其中：

- 第一列 代表 **教师职工编号**
- 第二列 代表 **教师姓名**
- 第三列 代表 **密码**

效果图：



在Login函数的教师分支中加入如下代码，验证教师身份

```
//教师登录验证
int fId;
string fName;
string fPwd;
while (ifs >> fId && ifs >> fName && ifs >> fPwd)
{
    if (id == fId && name == fName && pwd == fPwd)
    {
        cout << "教师验证登录成功!" << endl;
        system("pause");
        system("cls");
        person = new Teacher(id, name, pwd);
        return;
    }
}
```

添加代码效果图

```
if (type == 1)
{
    //学生登录验证
    int fId;
    string fName;
    string fPwd;
    while (ifs >> fId && ifs >> fName && ifs >> fPwd)
    {
        if (id == fId && name == fName && pwd == fPwd)
        {
            cout << "学生验证登录成功!" << endl;
            system("pause");
            system("cls");
            person = new Student(id, name, pwd);

            return;
        }
    }
}

else if (type == 2)
{
    //教师登录验证
    int fId;
    string fName;
    string fPwd;
    while (ifs >> fId && ifs >> fName && ifs >> fPwd)
    {
        if (id == fId && name == fName && pwd == fPwd)
        {
            cout << "教师验证登录成功!" << endl;
            system("pause");
            system("cls");
            person = new Teacher(id, name, pwd);
            return;
        }
    }
}
```

测试：



6.5 管理员登录实现

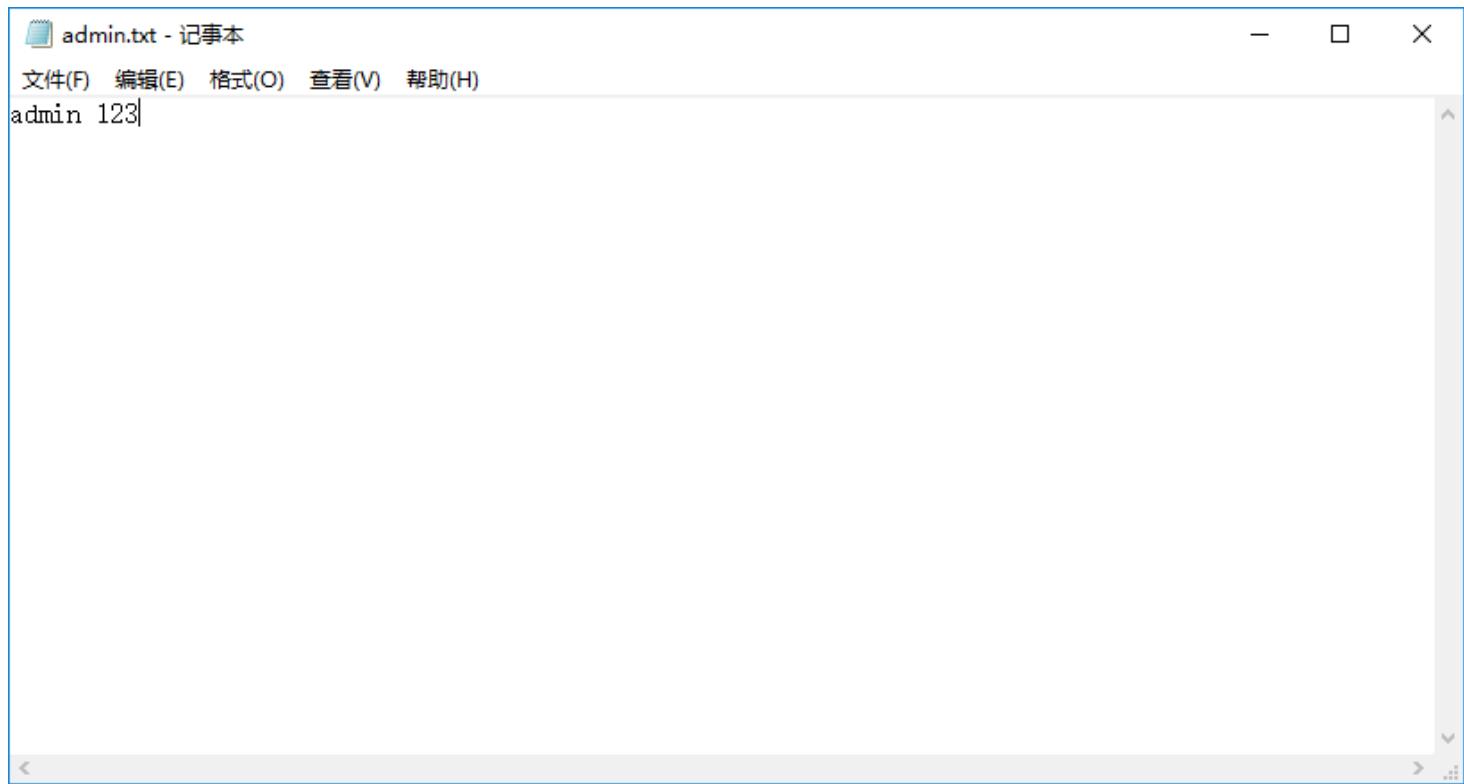
在admin.txt文件中添加一条管理员信息，由于我们只有一条管理员，因此本案例中没有添加管理员的功能

添加信息:

```
admin 123
```

其中: admin 代表管理员用户名， 123 代表管理员密码

效果图:



在Login函数的管理员分支中加入如下代码，验证管理员身份

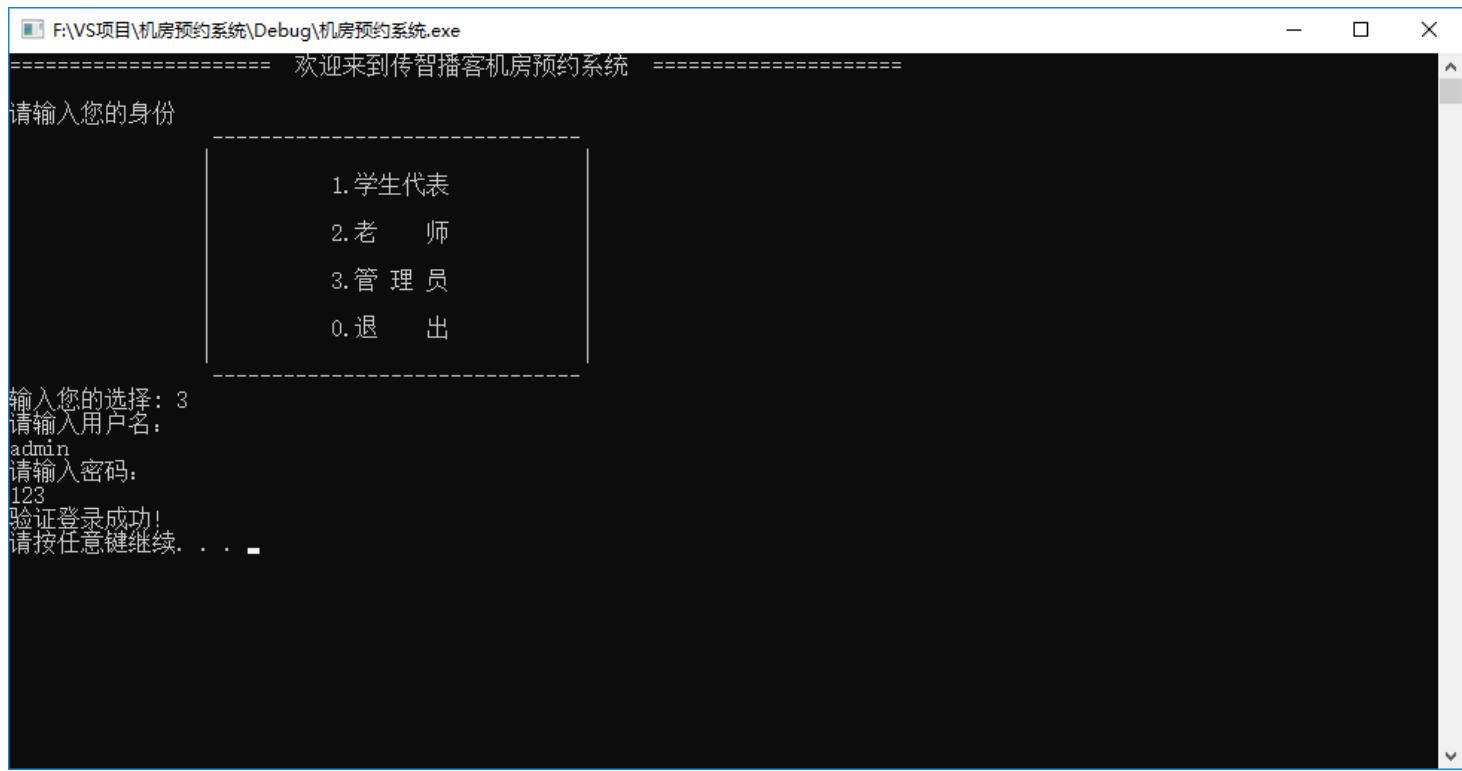
```
//管理员登录验证
    string fName;
    string fPwd;
    while (ifs >> fName && ifs >> fPwd)
    {
        if (name == fName && pwd == fPwd)
        {
            cout << "验证登录成功!" << endl;
            //登录成功后，按任意键进入管理员界面
            system("pause");
            system("cls");
            //创建管理员对象
            person = new Manager(name, pwd);
            return;
        }
    }
```

添加效果如图：

```
else if (type == 2)
{
    //教师登录验证
    int fId;
    string fName;
    string fPwd;
    while (ifs >> fId && ifs >> fName && ifs >> fPwd)
    {
        if (id == fId && name == fName && pwd == fPwd)
        {
            cout << "教师验证登录成功!" << endl;
            system("pause");
            system("cls");
            person = new Teacher(id, name, pwd);
            return;
        }
    }
}

else if(type == 3)
{
    //管理员登录验证
    string fName;
    string fPwd;
    while (ifs >> fName && ifs >> fPwd)
    {
        if (name == fName && pwd == fPwd)
        {
            cout << "管理员验证登录成功!" << endl;
            //登录成功后，按任意键进入管理员界面
            system("pause");
            system("cls");
            //创建管理员对象
            person = new Manager(name, pwd);
            return;
        }
    }
}
```

测试效果如图：



至此，所有身份的登录功能全部实现！

7、管理员模块

7.1 管理员登录和注销

7.1.1 构造函数

- 在Manager类的构造函数中，初始化管理员信息，代码如下：

```
//有参构造
Manager::Manager(string name, string pwd)
{
    this->m_Name = name;
    this->m_Pwd = pwd;
}
```

7.1.2 管理员子菜单

- 在机房预约系统.cpp中，当用户登录的是管理员，添加管理员菜单接口
- 将不同的分支提供出来
 - 添加账号
 - 查看账号
 - 查看机房
 - 清空预约
 - 注销登录

- 实现注销功能

添加全局函数 `void managerMenu(Identity * &manager)`，代码如下：

```
//管理员菜单
void managerMenu(Identity * &manager)
{
    while (true)
    {
        //管理员菜单
        manager->operMenu();

        Manager* man = (Manager*)manager;
        int select = 0;

        cin >> select;

        if (select == 1) //添加账号
        {
            cout << "添加账号" << endl;
            man->addPerson();
        }
        else if (select == 2) //查看账号
        {
            cout << "查看账号" << endl;
            man->showPerson();
        }
        else if (select == 3) //查看机房
        {
            cout << "查看机房" << endl;
            man->showComputer();
        }
        else if (select == 4) //清空预约
        {
            cout << "清空预约" << endl;
            man->cleanFile();
        }
        else
        {
            delete manager;
            cout << "注销成功" << endl;
            system("pause");
            system("cls");
            return;
        }
    }
}
```

7.1.3 菜单功能实现

- 在实现成员函数 `void Manager::operMenu()` 代码如下：

```
//选择菜单
void Manager::operMenu()
{
    cout << "欢迎管理员: "<<this->m_Name << "登录! " << endl;
    cout << "\t\t-----\n";
    cout << "\t\t|          |\n";
    cout << "\t\t|  1. 添加账号  |\n";
    cout << "\t\t|          |\n";
    cout << "\t\t|  2. 查看账号  |\n";
    cout << "\t\t|          |\n";
    cout << "\t\t|  3. 查看机房  |\n";
    cout << "\t\t|          |\n";
    cout << "\t\t|  4. 清空预约  |\n";
    cout << "\t\t|          |\n";
    cout << "\t\t|  0. 注销登录  |\n";
    cout << "\t\t|          |\n";
    cout << "\t\t-----\n";
    cout << "请选择您的操作: " << endl;
}
```

7.1.4 接口对接

- 管理员成功登录后，调用管理员子菜单界面
- 在管理员登录验证分支中，添加代码：

```
//进入管理员子菜单
managerMenu(person);
```

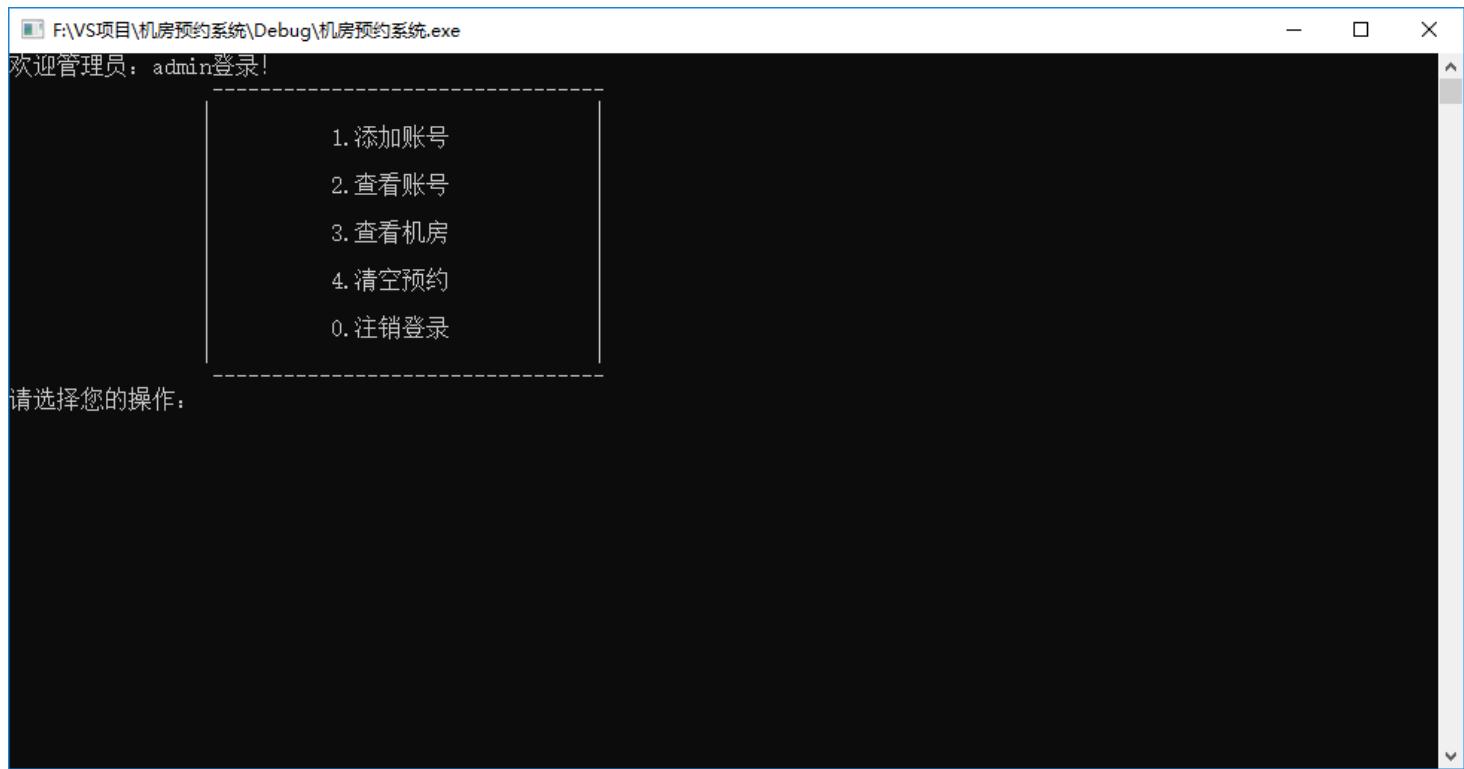
添加效果如：

```
else if(type == 3)
{
    //管理员登录验证
    string fName;
    string fPwd;
    while (ifs >> fName && ifs >> fPwd)
    {
        if (name == fName && pwd == fPwd)
        {
            cout << "管理员验证登录成功!" << endl;
            //登录成功后，按任意键进入管理员界面
            system("pause");
            system("cls");
            //创建管理员对象
            person = new Manager(name, pwd);
            //进入管理员子菜单
            managerMenu(person);
        }
        return;
    }
}
```

测试对接，效果如图：



登录成功



注销登录:



至此，管理员身份可以成功登录以及注销

7.2 添加账号

功能描述:

- 给学生或教师添加新的账号

功能要求：

- 添加时学生学号不能重复、教职工工号不能重复

7.2.1 添加功能实现

在Manager的**addPerson**成员函数中，实现添加新账号功能，代码如下：

```
//添加账号
void Manager::addPerson()
{
    cout << "请输入添加账号的类型" << endl;
    cout << "1、添加学生" << endl;
    cout << "2、添加老师" << endl;

    string fileName;
    string tip;
    ofstream ofs;

    int select = 0;
    cin >> select;

    if (select == 1)
    {
        fileName = STUDENT_FILE;
        tip = "请输入学号： ";
    }
    else
    {
        fileName = TEACHER_FILE;
        tip = "请输入职工编号： ";
    }

    ofs.open(fileName, ios::out | ios::app);
    int id;
    string name;
    string pwd;
    cout << tip << endl;
    cin >> id;

    cout << "请输入姓名：" << endl;
    cin >> name;

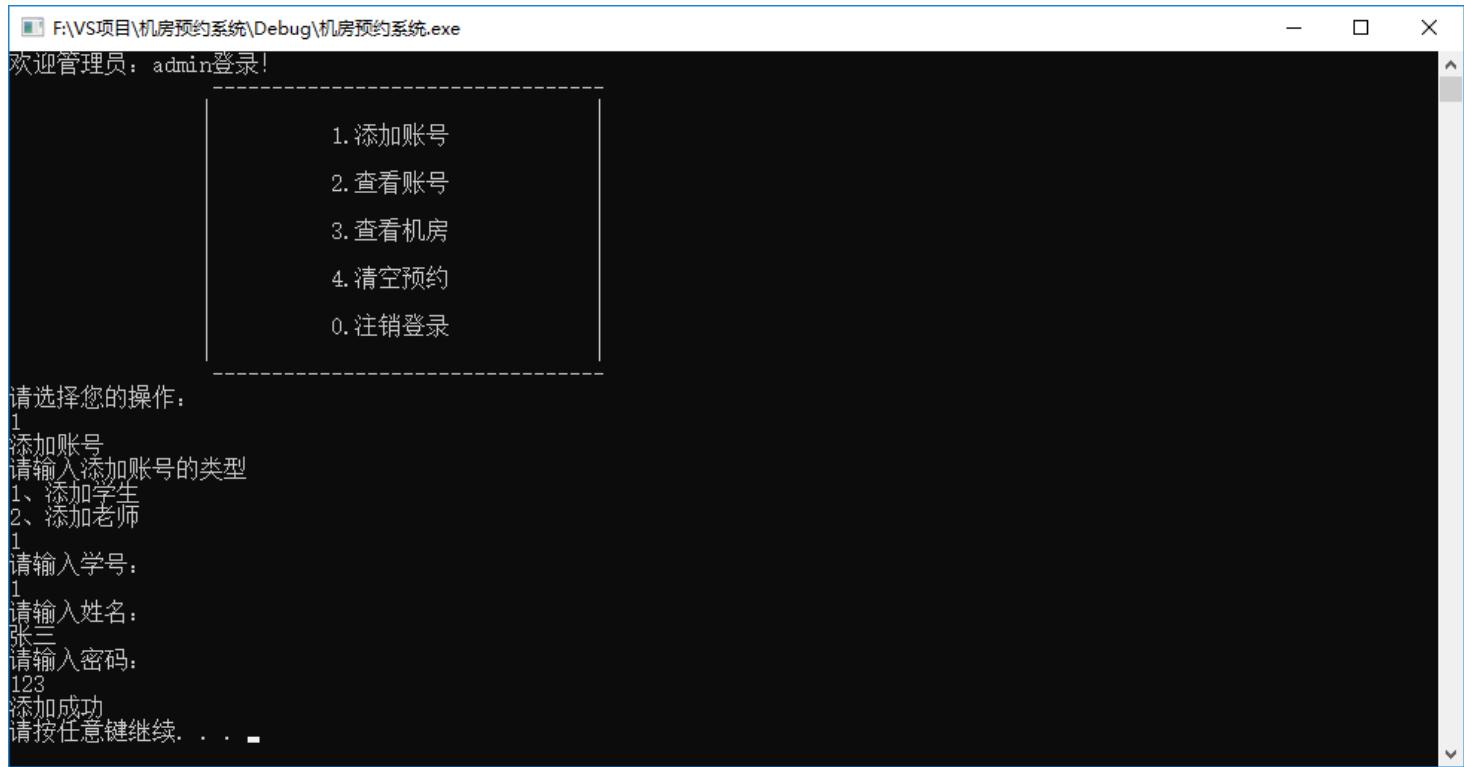
    cout << "请输入密码：" << endl;
    cin >> pwd;

    ofs << id << " " << name << " " << pwd << " " << endl;
    cout << "添加成功" << endl;

    system("pause");
    system("cls");

    ofs.close();
}
```

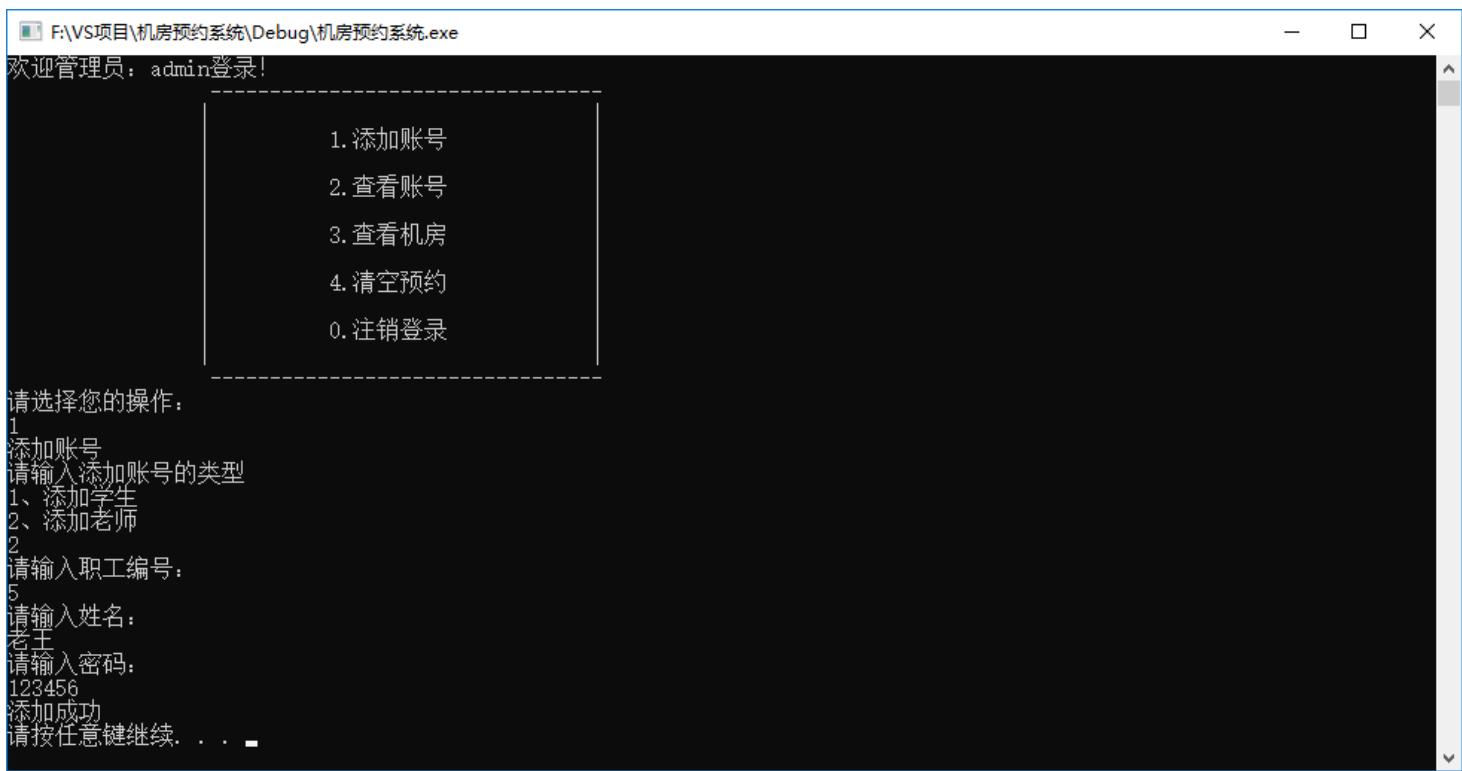
测试添加学生：



成功在学生文件中添加了一条信息



测试添加教师:



成功在教师文件中添加了一条信息



7.2.2 去重操作

功能描述：添加新账号时，如果是重复的学生编号，或是重复的教师职工编号，提示有误

7.2.2.1 读取信息

- 要去除重复的账号，首先要先将学生和教师的账号信息获取到程序中，方可检测

- 在manager.h中，添加两个容器，用于存放学生和教师的信息
- 添加一个新的成员函数 `void initVector()` 初始化容器

```
//初始化容器  
void initVector();  
  
//学生容器  
vector<Student> vStu;  
  
//教师容器  
vector<Teacher> vTea;
```

添加位置如图：

```
class Manager :public Identity
{
public:

    //默认构造
    Manager();

    //有参构造
    Manager(string name, string pwd);

    //选择菜单
    virtual void operMenu();

    //添加账号
    void addPerson();

    //查看账号
    void showPerson();

    //查看机房信息
    void showComputer();

    //清空预约记录
    void cleanFile();

    //初始化容器
    void initVector();

    //学生容器
    vector<Student> vStu;

    //教师容器
    vector<Teacher> vTea;
};

};
```

在Manager的有参构造函数中，获取目前的学生和教师信息

代码如下：

```

void Manager::initVector()
{
    //读取学生文件中信息
    ifstream ifs;
    ifs.open(STUDENT_FILE, ios::in);
    if (!ifs.is_open())
    {
        cout << "文件读取失败" << endl;
        return;
    }

    vStu.clear();
    vTea.clear();

    Student s;
    while (ifs >> s.m_Id && ifs >> s.m_Name && ifs >> s.m_Pwd)
    {
        vStu.push_back(s);
    }
    cout << "当前学生数量为: " << vStu.size() << endl;
    ifs.close(); //学生初始化

    //读取老师文件信息
    ifs.open(TEACHER_FILE, ios::in);

    Teacher t;
    while (ifs >> t.m_EmpId && ifs >> t.m_Name && ifs >> t.m_Pwd)
    {
        vTea.push_back(t);
    }
    cout << "当前教师数量为: " << vTea.size() << endl;

    ifs.close();
}

```

在有参构造函数中，调用初始化容器函数

```

//有参构造
Manager::Manager(string name, string pwd)
{
    this->m_Name = name;
    this->m_Pwd = pwd;

    //初始化容器
    this->initVector();
}

```

测试，运行代码可以看到测试代码获取当前学生和教师数量



7.2.2.2 去重函数封装

在manager.h文件中添加成员函数 `bool checkRepeat(int id, int type);`

```
//检测重复 参数:(传入id, 传入类型) 返回值: (true 代表有重复, false代表没有重复)
bool checkRepeat(int id, int type);
```

在manager.cpp文件中实现成员函数 `bool checkRepeat(int id, int type);`

```
bool Manager::checkRepeat(int id, int type)
{
    if (type == 1)
    {
        for (vector<Student>::iterator it = vStu.begin(); it != vStu.end(); it++)
        {
            if (id == it->m_Id)
            {
                return true;
            }
        }
    }
    else
    {
        for (vector<Teacher>::iterator it = vTea.begin(); it != vTea.end(); it++)
        {
            if (id == it->m_EmpId)
            {
                return true;
            }
        }
    }
    return false;
}
```

7.2.2.3 添加去重操作

在添加学生编号或者教师职工号时，检测是否有重复，代码如下：

```
string errorTip; //重复错误提示

if (select == 1)
{
    fileName = STUDENT_FILE;
    tip = "请输入学号：";
    errorTip = "学号重复，请重新输入";
}
else
{
    fileName = TEACHER_FILE;
    tip = "请输入职工编号：";
    errorTip = "职工号重复，请重新输入";
}
ofs.open(fileName, ios::out | ios::app);
int id;
string name;
string pwd;
cout << tip << endl;

while (true)
{
    cin >> id;

    bool ret = this->checkRepeat(id, 1);

    if (ret) //有重复
    {
        cout << errorTip << endl;
    }
    else
    {
        break;
    }
}
```

代码位置如图：

```
string errorTip; //重复错误提示

if (select == 1)
{
    fileName = STUDENT_FILE;
    tip = "请输入学号: ";
    errorTip = "学号重复, 请重新输入";
}
else
{
    fileName = TEACHER_FILE;
    tip = "请输入职工编号: ";
    errorTip = "职工号重复, 请重新输入";
}

ofs.open(fileName, ios::out | ios::app);
int id;
string name;
string pwd;
cout << tip << endl;

while (true)
{
    cin >> id;

    bool ret = this->checkRepeat(id, 1);

    if (ret) //有重复
    {
        cout << errorTip << endl;
    }
    else
    {
        break;
    }
}
```

检测效果：



7.2.2.4 bug解决

bug描述：

- 虽然可以检测重复的账号，但是刚添加的账号由于没有更新到容器中，因此不会做检测
- 导致刚加入的账号的学生号或者职工编号，再次添加时依然可以重复

解决方案：

- 在每次添加新账号时，重新初始化容器

在添加完毕后，加入代码：

```
//初始化容器  
this->initVector();
```

位置如图：

```
cout << "请输入姓名: " << endl;
cin >> name;

cout << "请输入密码: " << endl;
cin >> pwd;

ofs << id << " " << name << " " << pwd << " " << endl;
cout << "添加成功" << endl;

system("pause");
system("cls");

ofs.close();

//初始化容器
this->initVector();
}
```

再次测试，刚加入的账号不会重复添加了！

7.3 显示账号

功能描述：显示学生信息或教师信息

7.3.1 显示功能实现

在Manager的**showPerson**成员函数中，实现显示账号功能，代码如下：

```
void printStudent(Student & s)
{
    cout << "学号: " << s.m_Id << " 姓名: " << s.m_Name << " 密码: " << s.m_Pwd << endl;
}
void printTeacher(Teacher & t)
{
    cout << "职工号: " << t.m_EmpId << " 姓名: " << t.m_Name << " 密码: " << t.m_Pwd << endl
}

void Manager::showPerson()
{
    cout << "请选择查看内容: " << endl;
    cout << "1、查看所有学生" << endl;
    cout << "2、查看所有老师" << endl;

    int select = 0;

    cin >> select;

    if (select == 1)
    {
        cout << "所有学生信息如下: " << endl;
        for_each(vStu.begin(), vStu.end(), printStudent);
    }
    else
    {
        cout << "所有老师信息如下: " << endl;
        for_each(vTea.begin(), vTea.end(), printTeacher);
    }
    system("pause");
    system("cls");
}
```

7.3.2 测试

测试查看学生效果

```
F:\VS项目\机房预约系统\Debug\机房预约系统.exe
当前学生数量为: 3
当前教师数量为: 1
欢迎管理员: admin登录!
1. 添加账号
2. 查看账号
3. 查看机房
4. 清空预约
0. 注销登录

请选择您的操作:
2
查看账号
请选择查看内容:
1、查看所有学生
2、查看所有老师
1
所有学生信息如下:
学号: 1 姓名: 张三 密码: 123
学号: 2 姓名: 李四 密码: 123456
学号: 3 姓名: 王五 密码: 111
请按任意键继续. . .
```

测试查看教师效果

```
F:\VS项目\机房预约系统\Debug\机房预约系统.exe
欢迎管理员: admin登录!
1. 添加账号
2. 查看账号
3. 查看机房
4. 清空预约
0. 注销登录

请选择您的操作:
2
查看账号
请选择查看内容:
1、查看所有学生
2、查看所有老师
2
所有老师信息如下:
职工号: 5 姓名: 老王 密码: 123456
请按任意键继续. . .
```

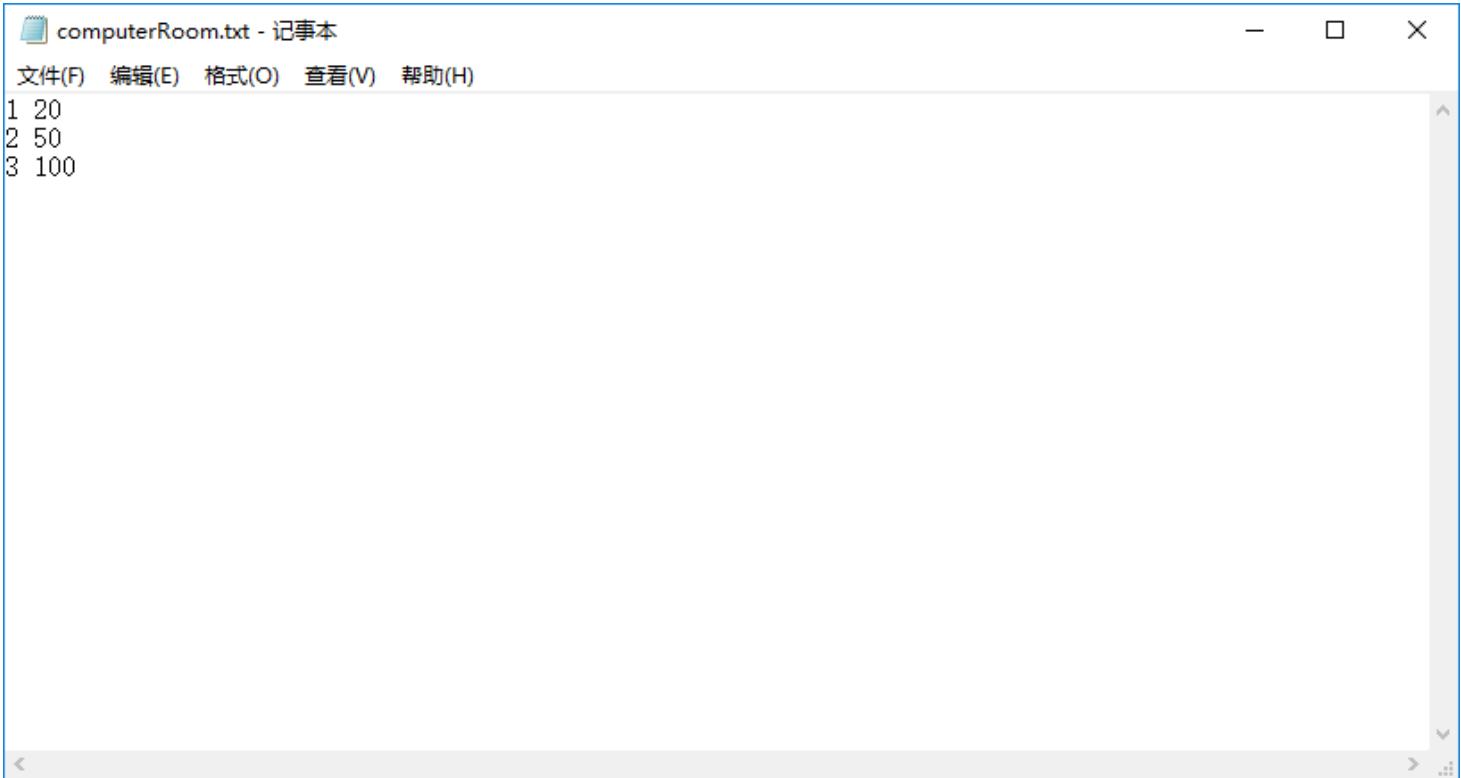
至此，显示账号功能实现完毕

7.4 查看机房

7.4.1 添加机房信息

案例需求中，机房一共有三个，其中1号机房容量20台机器，2号50台，3号100台

我们可以将信息录入到computerRoom.txt中



7.4.2 机房类创建

在头文件下，创建新的文件 computerRoom.h

并添加如下代码：

```
#pragma once
#include<iostream>
using namespace std;
//机房类
class ComputerRoom
{
public:
    int m_CoId; //机房id号
    int m_MaxNum; //机房最大容量
};
```

7.4.3 初始化机房信息

在Manager管理员类下，添加机房的容器,用于保存机房信息

```
//机房容器
vector<ComputerRoom> vCom;
```

在Manager有参构造函数中，追加如下代码，初始化机房信息

```
//获取机房信息
ifstream ifs;

ifs.open(COMPUTER_FILE, ios::in);

ComputerRoom c;
while (ifs >> c.m_ComId && ifs >> c.m_MaxNum)
{
    vCom.push_back(c);
}
cout << "当前机房数量为: " << vCom.size() << endl;

ifs.close();
```

位置如图：

```
//有参构造
Manager::Manager(string name, string pwd)
{
    this->m_Name = name;
    this->m_Pwd = pwd;
    //初始化容器
    this->initVector();

//获取机房信息
ifstream ifs;

ifs.open(COMPUTER_FILE, ios::in);

ComputerRoom c;
while (ifs >> c.m_ComId && ifs >> c.m_MaxNum)
{
    vCom.push_back(c);
}
cout << "当前机房数量为: " << vCom.size() << endl;

ifs.close();
}
```

因为机房信息目前版本不会有所改动，如果后期有修改功能，最好封装到一个函数中，方便维护

7.4.4 显示机房信息

在Manager类的showComputer成员函数中添加如下代码：

```
//查看机房信息
void Manager::showComputer()
{
    cout << "机房信息如下: " << endl;
    for (vector<ComputerRoom>::iterator it = vCom.begin(); it != vCom.end(); it++)
    {
        cout << "机房编号: " << it->m_ComId << " 机房最大容量: " << it->m_MaxNum << endl
    }
    system("pause");
    system("cls");
}
```

测试显示机房信息功能：



7.5 清空预约

功能描述：

清空生成的 order.txt 预约文件

7.5.1 清空功能实现

在Manager的cleanFile成员函数中添加如下代码：

```
//清空预约记录
void Manager::cleanFile()
{
    ofstream ofs(ORDER_FILE, ios::trunc);
    ofs.close();

    cout << "清空成功! " << endl;
    system("pause");
    system("cls");
}
```

测试清空，可以随意写入一些信息在order.txt中，然后调用cleanFile清空文件接口，查看是否清空干净

8、 学生模块

8.1 学生登录和注销

8.1.1 构造函数

- 在Student类的构造函数中，初始化学生信息，代码如下：

```
//有参构造(学号、姓名、密码)
Student::Student(int id, string name, string pwd)
{
    //初始化属性
    this->m_Id = id;
    this->m_Name = name;
    this->m_Pwd = pwd;
}
```

8.1.2 管理员子菜单

- 在机房预约系统.cpp中，当用户登录的是学生，添加学生菜单接口
- 将不同的分支提供出来
 - 申请预约
 - 查看我的预约
 - 查看所有预约
 - 取消预约
 - 注销登录
- 实现注销功能

添加全局函数 void studentMenu(Identity * &manager) 代码如下：

```

//学生菜单
void studentMenu(Identity * &student)
{
    while (true)
    {
        //学生菜单
        student->operMenu();

        Student* stu = (Student*)student;
        int select = 0;

        cin >> select;

        if (select == 1) //申请预约
        {
            stu->applyOrder();
        }
        else if (select == 2) //查看自身预约
        {
            stu->showMyOrder();
        }
        else if (select == 3) //查看所有预约
        {
            stu->showAllOrder();
        }
        else if (select == 4) //取消预约
        {
            stu->cancelOrder();
        }
        else
        {
            delete student;
            cout << "注销成功" << endl;
            system("pause");
            system("cls");
            return;
        }
    }
}

```

8.1.3 菜单功能实现

- 在实现成员函数 void Student::operMenu() 代码如下：

```
//菜单界面
void Student::operMenu()
{
    cout << "欢迎学生代表：" << this->m_Name << "登录！" << endl;
    cout << "\t\t-----\n";
    cout << "\t\t|      |\n";
    cout << "\t\t|  1. 申请预约  |\n";
    cout << "\t\t|  2. 查看我的预约  |\n";
    cout << "\t\t|  3. 查看所有预约  |\n";
    cout << "\t\t|  4. 取消预约  |\n";
    cout << "\t\t|  0. 注销登录  |\n";
    cout << "\t\t-----\n";
    cout << "请选择您的操作： " << endl;
}
```

8.1.4 接口对接

- 学生成功登录后，调用学生的子菜单界面
- 在学生登录分支中，添加代码：

```
//进入学生子菜单
studentMenu(person);
```

添加效果如图：

```
if (type == 1)
{
    //学生登录验证
    int fId;
    string fName;
    string fPwd;
    while (ifs >> fId && ifs >> fName && ifs >> fPwd)
    {
        if (id == fId && name == fName && pwd == fPwd)
        {
            cout << "学生验证登录成功!" << endl;
            system("pause");
            system("cls");
            person = new Student(id, name, pwd);
            //进入学生子菜单
            studentMenu(person);
            return;
        }
    }
}
else if (type == 2)
{
    //教师登录验证
    int fId;
    string fName;
    string fPwd;
    while (ifs >> fId && ifs >> fName && ifs >> fPwd)
    {
        if (id == fId && name == fName && pwd == fPwd)
        {
            cout << "教师验证登录成功!" << endl;
            system("pause");
            system("cls");
            person = new Teacher(id, name, pwd);
            return;
        }
    }
}
```

测试对接，效果如图：

登录验证通过：



学生子菜单:



注销登录:



8.2 申请预约

8.2.1 获取机房信息

- 在申请预约时，学生可以看到机房的信息，因此我们需要让学生获取到机房的信息

在student.h中添加新的成员函数如下：

```
//机房容器
vector<ComputerRoom> vCom;
```

在学生的有参构造函数中追加如下代码：

```
//获取机房信息
ifstream ifs;
ifs.open(COMPUTER_FILE, ios::in);

ComputerRoom c;
while (ifs >> c.m_CoId && ifs >> c.m_MaxNum)
{
    vCom.push_back(c);
}

ifs.close();
```

追加位置如图：

```
//有参构造(学号、姓名、密码)
Student::Student(int id, string name, string pwd)
{
    //初始化属性
    this->m_Id = id;
    this->m_Name = name;
    this->m_Pwd = pwd;

    //获取机房信息
    ifstream ifs;
    ifs.open(COMPUTER_FILE, ios::in);

    ComputerRoom c;
    while (ifs >> c.m_ComId && ifs >> c.m_MaxNum)
    {
        vCom.push_back(c);
    }

    ifs.close();
}
```

至此，vCom容器中保存了所有机房的信息

8.2.2 预约功能实现

在student.cpp中实现成员函数 void Student::applyOrder()

```
//申请预约
void Student::applyOrder()
{
    cout << "机房开放时间为周一至周五！" << endl;
    cout << "请输入申请预约的时间：" << endl;
    cout << "1、周一" << endl;
    cout << "2、周二" << endl;
    cout << "3、周三" << endl;
    cout << "4、周四" << endl;
    cout << "5、周五" << endl;
    int date = 0;
    int interval = 0;
    int room = 0;

    while (true)
    {
        cin >> date;
        if (date >= 1 && date <= 5)
        {
            break;
        }
        cout << "输入有误，请重新输入" << endl;
    }

    cout << "请输入申请预约的时间段：" << endl;
    cout << "1、上午" << endl;
    cout << "2、下午" << endl;

    while (true)
    {
        cin >> interval;
        if (interval >= 1 && interval <= 2)
        {
            break;
        }
        cout << "输入有误，请重新输入" << endl;
    }

    cout << "请选择机房：" << endl;
    cout << "1号机房容量：" << vCom[0].m_MaxNum << endl;
    cout << "2号机房容量：" << vCom[1].m_MaxNum << endl;
    cout << "3号机房容量：" << vCom[2].m_MaxNum << endl;

    while (true)
    {
        cin >> room;
        if (room >= 1 && room <= 3)
        {
            break;
        }
        cout << "输入有误，请重新输入" << endl;
    }

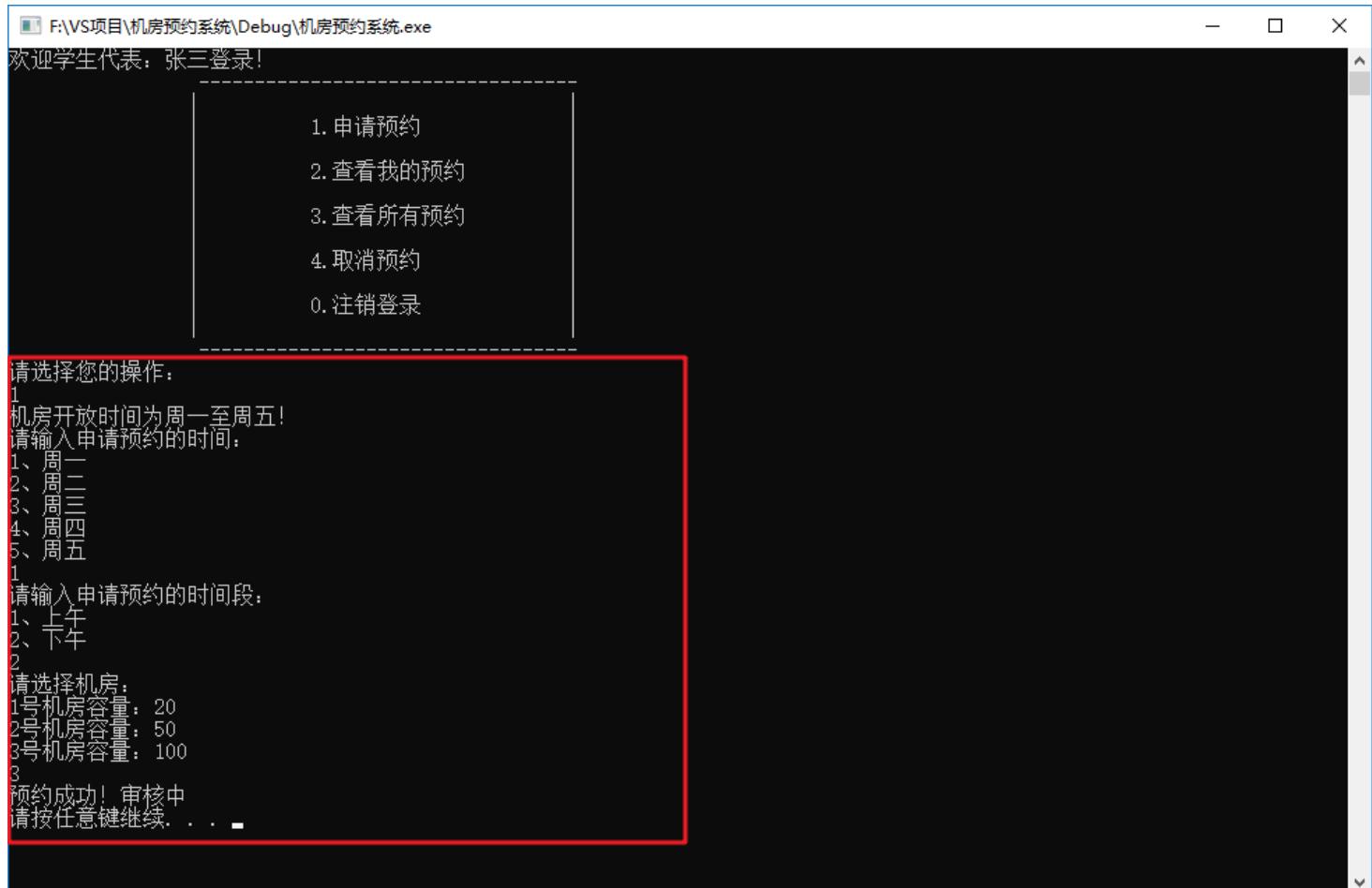
    cout << "预约成功！审核中" << endl;
```

```
ofstream ofs(ORDER_FILE, ios::app);
ofs << "date:" << date << " ";
ofs << "interval:" << interval << " ";
ofs << "stuId:" << this->m_Id << " ";
ofs << "stuName:" << this->m_Name << " ";
ofs << "roomId:" << room << " ";
ofs << "status:" << 1 << endl;

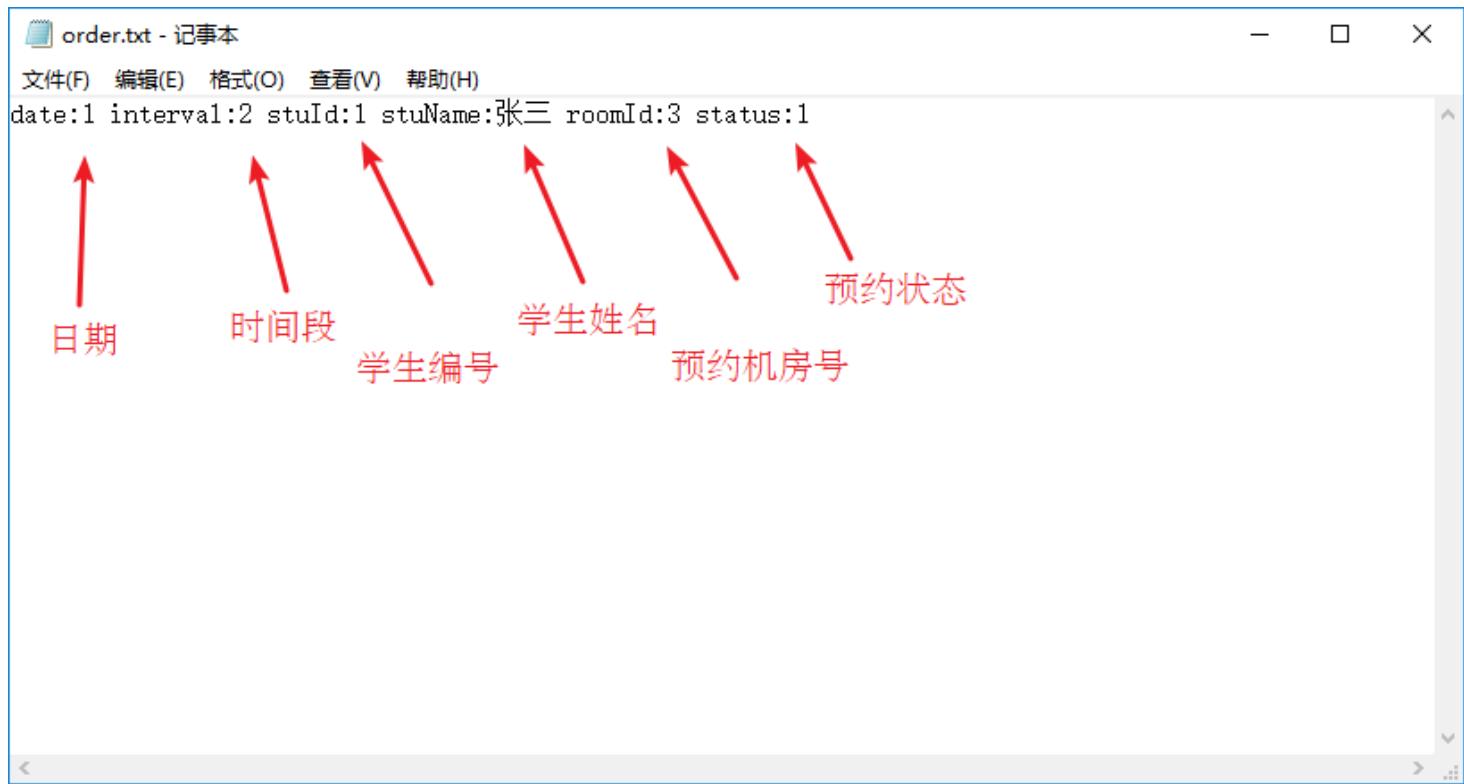
ofs.close();

system("pause");
system("cls");
}
```

运行程序，测试代码：



在order.txt文件中生成如下内容：



8.3 显示预约

8.3.1 创建预约类

功能描述：显示预约记录时，需要从文件中获取到所有记录，用来显示，创建预约的类来管理记录以及更新

在头文件以及源文件下分别创建**orderFile.h** 和 **orderFile.cpp**文件

orderFile.h中添加如下代码：

```
#pragma once
#include<iostream>
using namespace std;
#include <map>
#include "globalFile.h"

class OrderFile
{
public:

    //构造函数
    OrderFile();

    //更新预约记录
    void updateOrder();

    //记录的容器 key --- 记录的条数 value --- 具体记录的键值对信息
    map<int, map<string, string>> m_orderData;

    //预约记录条数
    int m_Size;
};

};
```

构造函数中获取所有信息，并存放在容器中，添加如下代码：

```
OrderFile::OrderFile()
{
    ifstream ifs;
    ifs.open(ORDER_FILE, ios::in);

    string date;      //日期
    string interval; //时间段
    string stuId;    //学生编号
    string stuName;  //学生姓名
    string roomId;   //机房编号
    string status;   //预约状态

    this->m_Size = 0; //预约记录个数

    while (ifs >> date && ifs >> interval && ifs >> stuId && ifs >> stuName && ifs >> roomId)
    {
        //测试代码
        /*
        cout << date << endl;
        cout << interval << endl;
        cout << stuId << endl;
        cout << stuName << endl;
        cout << roomId << endl;
        cout << status << endl;
        */

        string key;
        string value;
        map<string, string> m;

        int pos = date.find(":");
        if (pos != -1)
        {
            key = date.substr(0, pos);
            value = date.substr(pos + 1, date.size() - pos - 1);
            m.insert(make_pair(key, value));
        }

        pos = interval.find(":");
        if (pos != -1)
        {
            key = interval.substr(0, pos);
            value = interval.substr(pos + 1, interval.size() - pos - 1 );
            m.insert(make_pair(key, value));
        }

        pos = stuId.find(":");
        if (pos != -1)
        {
            key = stuId.substr(0, pos);
            value = stuId.substr(pos + 1, stuId.size() - pos - 1 );
            m.insert(make_pair(key, value));
        }
    }
}
```

```

pos = stuName.find(":");
if (pos != -1)
{
    key = stuName.substr(0, pos);
    value = stuName.substr(pos + 1, stuName.size() - pos -1);
    m.insert(make_pair(key, value));
}

pos = roomId.find(":");
if (pos != -1)
{
    key = roomId.substr(0, pos);
    value = roomId.substr(pos + 1, roomId.size() - pos -1 );
    m.insert(make_pair(key, value));
}

pos = status.find(":");
if (pos != -1)
{
    key = status.substr(0, pos);
    value = status.substr(pos + 1, status.size() - pos -1);
    m.insert(make_pair(key, value));
}

this->m_orderData.insert(make_pair(this->m_Size, m));
this->m_Size++;
}

//测试代码
//for (map<int, map<string, string>>::iterator it = m_orderData.begin(); it != m_orderD
//{
//    cout << "key = " << it->first << " value = " << endl;
//    for (map<string, string>::iterator mit = it->second.begin(); mit != it->second.
//    {
//        cout << mit->first << " " << mit->second << " ";
//    }
//    cout << endl;
//}

ifs.close();
}

```

更新预约记录的成员函数updateOrder代码如下：

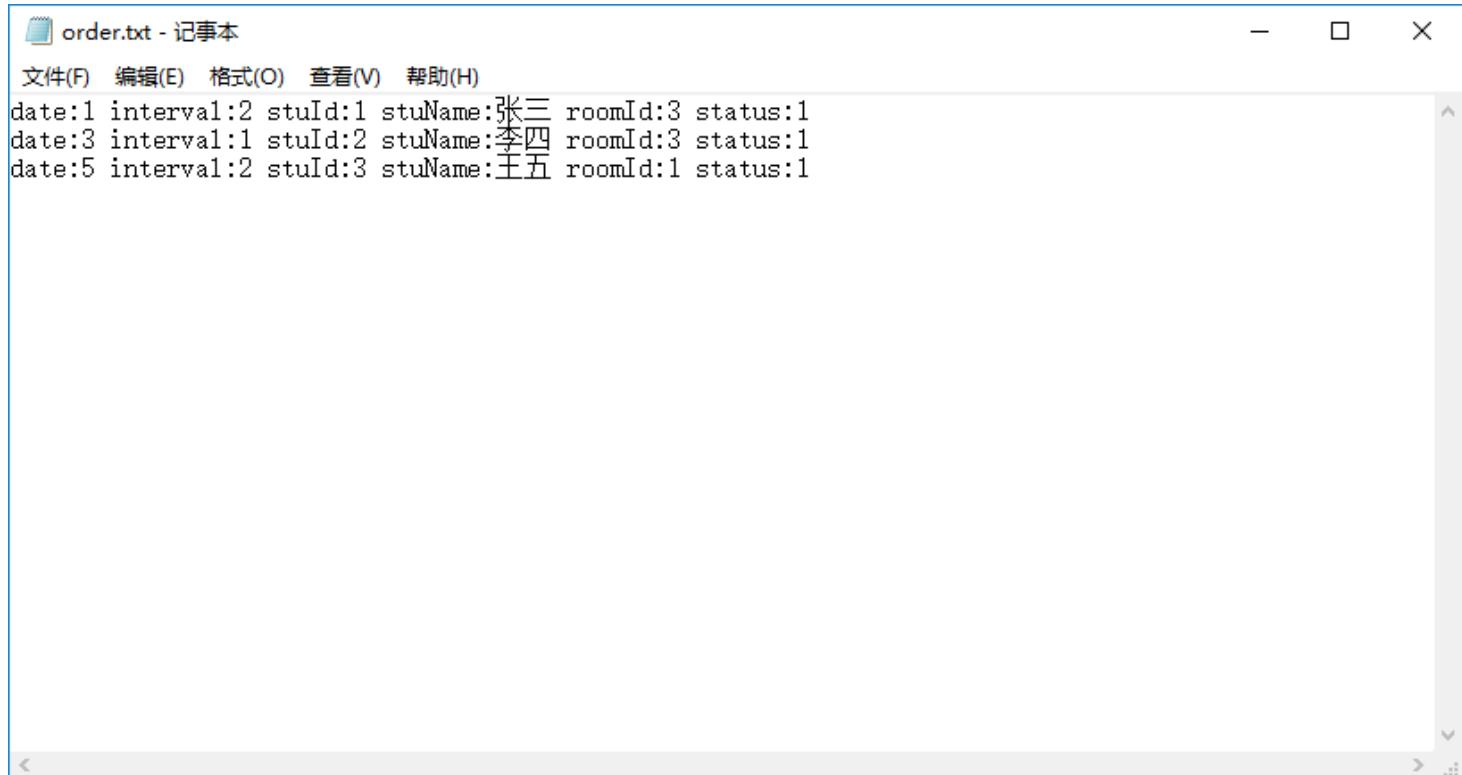
```
void OrderFile::updateOrder()
{
    if (this->m_Size == 0)
    {
        return;
    }

    ofstream ofs(ORDER_FILE, ios::out | ios::trunc);
    for (int i = 0; i < m_Size;i++)
    {
        ofs << "date:" << this->m_orderData[i]["date"] << " ";
        ofs << "interval:" << this->m_orderData[i]["interval"] << " ";
        ofs << "stuId:" << this->m_orderData[i]["stuId"] << " ";
        ofs << "stuName:" << this->m_orderData[i]["stuName"] << " ";
        ofs << "roomId:" << this->m_orderData[i]["roomId"] << " ";
        ofs << "status:" << this->m_orderData[i]["status"] << endl;
    }
    ofs.close();
}
```

8.3.2 显示自身预约

首先我们先添加几条预约记录，可以用程序添加或者直接修改order.txt文件

order.txt文件内容如下： 比如我们有三名同学分别产生了3条预约记录



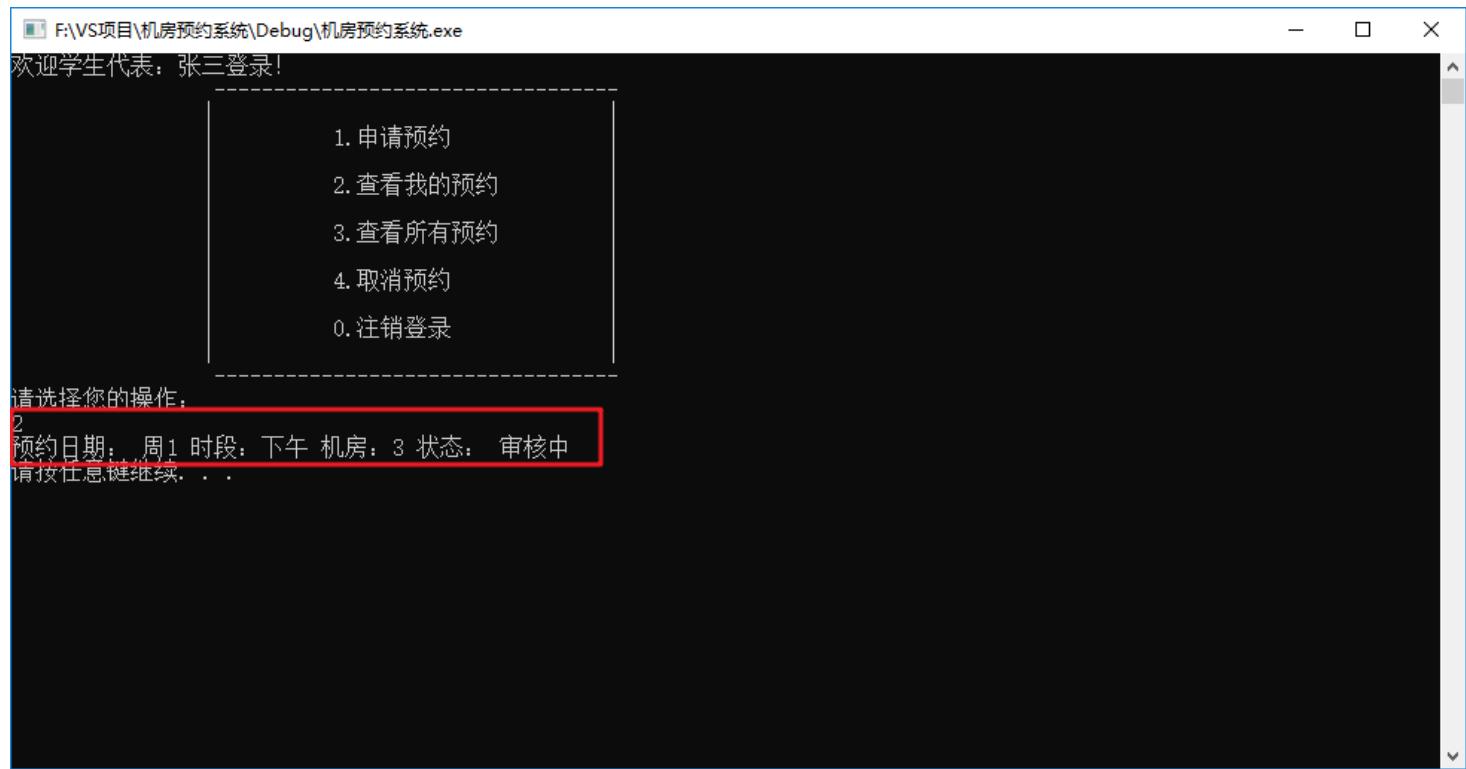
在Student类的 void Student::showMyOrder() 成员函数中，添加如下代码

```

//查看我的预约
void Student::showMyOrder()
{
    OrderFile of;
    if (of.m_Size == 0)
    {
        cout << "无预约记录" << endl;
        system("pause");
        system("cls");
        return;
    }
    for (int i = 0; i < of.m_Size; i++)
    {
        if (atoi(of.m_orderData[i]["stuId"].c_str()) == this->m_Id)
        {
            cout << "预约日期: 周" << of.m_orderData[i]["date"];
            cout << " 时段: " << (of.m_orderData[i]["interval"] == "1" ? "上午" : "下
            cout << " 机房: " << of.m_orderData[i]["roomId"];
            string status = " 状态: "; // 0 取消的预约 1 审核中 2 已预约 -1 预约失
            if (of.m_orderData[i]["status"] == "1")
            {
                status += "审核中";
            }
            else if (of.m_orderData[i]["status"] == "2")
            {
                status += "预约成功";
            }
            else if (of.m_orderData[i]["status"] == "-1")
            {
                status += "审核未通过，预约失败";
            }
            else
            {
                status += "预约已取消";
            }
            cout << status << endl;
        }
    }
    system("pause");
    system("cls");
}

```

测试效果如图：



8.3.3 显示所有预约

在Student类的 void Student::showAllOrder() 成员函数中，添加如下代码

```

//查看所有预约
void Student::showAllOrder()
{
    OrderFile of;
    if (of.m_Size == 0)
    {
        cout << "无预约记录" << endl;
        system("pause");
        system("cls");
        return;
    }

    for (int i = 0; i < of.m_Size; i++)
    {
        cout << i + 1 << "、 ";

        cout << "预约日期: 周" << of.m_orderData[i]["date"];
        cout << " 时段: " << (of.m_orderData[i]["interval"] == "1" ? "上午" : "下午");
        cout << " 学号: " << of.m_orderData[i]["stuId"];
        cout << " 姓名: " << of.m_orderData[i]["stuName"];
        cout << " 机房: " << of.m_orderData[i]["roomId"];
        string status = " 状态: "; // 0 取消的预约 1 审核中 2 已预约 -1 预约失败
        if (of.m_orderData[i]["status"] == "1")
        {
            status += "审核中";
        }
        else if (of.m_orderData[i]["status"] == "2")
        {
            status += "预约成功";
        }
        else if (of.m_orderData[i]["status"] == "-1")
        {
            status += "审核未通过，预约失败";
        }
        else
        {
            status += "预约已取消";
        }
        cout << status << endl;
    }

    system("pause");
    system("cls");
}

```

测试效果如图：



8.4 取消预约

在Student类的 void Student::cancelOrder() 成员函数中，添加如下代码

```

//取消预约
void Student::cancelOrder()
{
    OrderFile of;
    if (of.m_Size == 0)
    {
        cout << "无预约记录" << endl;
        system("pause");
        system("cls");
        return;
    }
    cout << "审核中或预约成功的记录可以取消, 请输入取消的记录" << endl;

    vector<int>v;
    int index = 1;
    for (int i = 0; i < of.m_Size; i++)
    {
        if (atoi(of.m_orderData[i]["stuId"].c_str()) == this->m_Id)
        {
            if (of.m_orderData[i]["status"] == "1" || of.m_orderData[i]["status"] =
            {
                v.push_back(i);
                cout << index ++ << "、 ";
                cout << "预约日期: 周" << of.m_orderData[i]["date"];
                cout << " 时段: " << (of.m_orderData[i]["interval"] == "1" ? "上
                cout << " 机房: " << of.m_orderData[i]["roomId"];
                string status = " 状态: ";// 0 取消的预约 1 审核中 2 已预约
                if (of.m_orderData[i]["status"] == "1")
                {
                    status += "审核中";
                }
                else if (of.m_orderData[i]["status"] == "2")
                {
                    status += "预约成功";
                }
                cout << status << endl;
            }
        }
    }
}

cout << "请输入取消的记录, 0代表返回" << endl;
int select = 0;
while (true)
{
    cin >> select;
    if (select >= 0 && select <= v.size())
    {
        if (select == 0)
        {
            break;
        }
        else
        {
            // cout << "记录所在位置: " << v[select - 1] << endl;
        }
    }
}

```

```
        of.m_orderData[v[select - 1]]["status"] = "0";
        of.updateOrder();
        cout << "已取消预约" << endl;
        break;
    }

}

cout << "输入有误, 请重新输入" << endl;
}

system("pause");
system("cls");
}
```

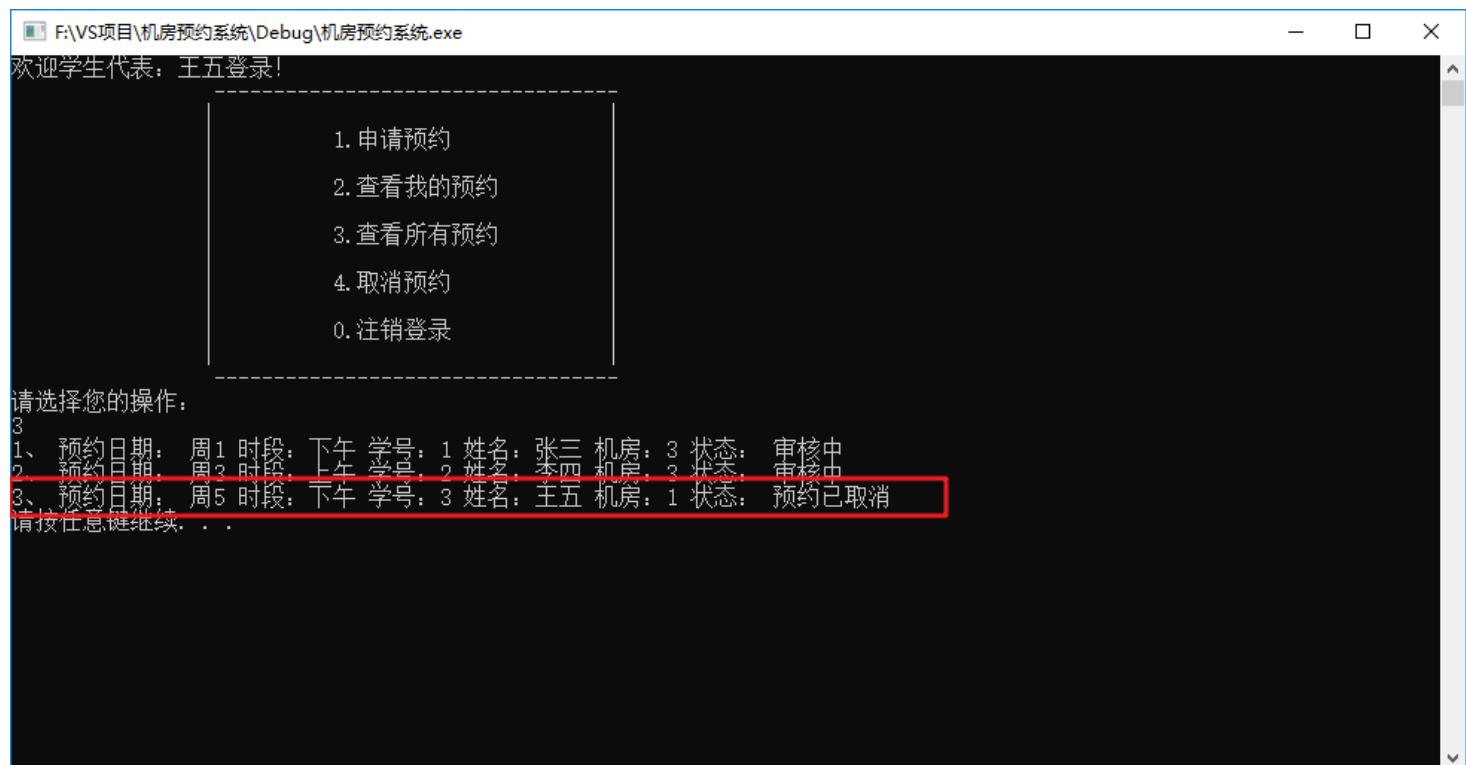
测试取消预约：



再次查看个人预约记录：



查看所有预约



查看order.txt预约文件

```
date:1 interval:2 stuId:1 stuName:张三 roomId:3 status:1
date:3 interval:1 stuId:2 stuName:李四 roomId:3 status:1
date:5 interval:2 stuId:3 stuName:王五 roomId:1 status:0
```

至此，学生模块功能全部实现

9、教师模块

9.1 教师登录和注销

9.1.1 构造函数

- 在Teacher类的构造函数中，初始化教师信息，代码如下：

```
//有参构造 (职工编号, 姓名, 密码)
Teacher::Teacher(int empId, string name, string pwd)
{
    //初始化属性
    this->m_EmpId = empId;
    this->m_Name = name;
    this->m_Pwd = pwd;
}
```

9.1.2 教师子菜单

- 在机房预约系统.cpp中，当用户登录的是教师，添加教师菜单接口
- 将不同的分支提供出来
 - 查看所有预约
 - 审核预约
 - 注销登录

- 实现注销功能

添加全局函数 `void TeacherMenu(Person * &manager)` 代码如下：

```
//教师菜单
void TeacherMenu(Identity * &teacher)
{
    while (true)
    {
        //教师菜单
        teacher->operMenu();

        Teacher* tea = (Teacher*)teacher;
        int select = 0;

        cin >> select;

        if (select == 1)
        {
            //查看所有预约
            tea->showAllOrder();
        }
        else if (select == 2)
        {
            //审核预约
            tea->validOrder();
        }
        else
        {
            delete teacher;
            cout << "注销成功" << endl;
            system("pause");
            system("cls");
            return;
        }
    }
}
```

9.1.3 菜单功能实现

- 在实现成员函数 `void Teacher::operMenu()` 代码如下：

```
//教师菜单界面
void Teacher::operMenu()
{
    cout << "欢迎教师: " << this->m_Name << "登录! " << endl;
    cout << "\t\t-----\n";
    cout << "\t\t|\n";
    cout << "\t\t| 1. 查看所有预约\n";
    cout << "\t\t| 2. 审核预约\n";
    cout << "\t\t| 0. 注销登录\n";
    cout << "\t\t|\n";
    cout << "\t\t-----\n";
    cout << "请选择您的操作: " << endl;
}
```

9.1.4 接口对接

- 教师成功登录后，调用教师的子菜单界面
- 在教师登录分支中，添加代码：

```
//进入教师子菜单
TeacherMenu(person);
```

添加效果如图：

orderFile.h student.cpp 机房预约系统.cpp ✘ globalFile.h computerRoom.h teacher.cpp*

全局范围 LoginIn(string fileName, int type)

```
192     else if (type == 2)
193     {
194         //教师登录验证
195         int fId;
196         string fName;
197         string fPwd;
198         while (ifs >> fId && ifs >> fName && ifs >> fPwd)
199         {
200             if (id == fId && name == fName && pwd == fPwd)
201             {
202                 cout << "教师验证登录成功!" << endl;
203                 system("pause");
204                 system("cls");
205                 person = new Teacher(id, name, pwd);
206                 //进入教师子菜单
207                 TeacherMenu(person);
208                 return;
209             }
210         }
211     }
212     else if(type == 3)
213     {
214         //管理员登录验证
215         string fName;
216         string fPwd;
217         while (ifs >> fName && ifs >> fPwd)
218         {
219             if (name == fName && pwd == fPwd)
220             {
221                 cout << "管理员验证登录成功!" << endl;
222                 //登录成功后，按任意键进入管理员界面
223                 system("pause");
224                 system("cls");
```

测试对接，效果如图：

登录验证通过：

```
F:\VS项目\机房预约系统\Debug\机房预约系统.exe
===== 欢迎来到传智播客机房预约系统 =====
请输入您的身份
1. 学生代表
2. 老师
3. 管理员
0. 退出

输入您的选择: 2
请输入你的职工号
5
请输入用户名:
老王
请输入密码:
123456
教师验证登录成功!
请按任意键继续. . .
```

教师子菜单：

```
F:\VS项目\机房预约系统\Debug\机房预约系统.exe
欢迎教师: 老王登录!
请输入您的操作:
1. 查看所有预约
2. 审核预约
0. 注销登录

请选择您的操作:
```

注销登录：



9.2 查看所有预约

9.2.1 所有预约功能实现

该功能与学生身份的查看所有预约功能相似，用于显示所有预约记录

在Teacher.cpp中实现成员函数 void Teacher::showAllOrder()

```

void Teacher::showAllOrder()
{
    OrderFile of;
    if (of.m_Size == 0)
    {
        cout << "无预约记录" << endl;
        system("pause");
        system("cls");
        return;
    }
    for (int i = 0; i < of.m_Size; i++)
    {
        cout << i + 1 << "、 ";
        cout << "预约日期: 周" << of.m_orderData[i]["date"];
        cout << " 时段: " << (of.m_orderData[i]["interval"] == "1" ? "上午" : "下午");
        cout << " 学号: " << of.m_orderData[i]["stuId"];
        cout << " 姓名: " << of.m_orderData[i]["stuName"];
        cout << " 机房: " << of.m_orderData[i]["roomId"];
        string status = " 状态: "; // 0 取消的预约 1 审核中 2 已预约 -1 预约失败
        if (of.m_orderData[i]["status"] == "1")
        {
            status += "审核中";
        }
        else if (of.m_orderData[i]["status"] == "2")
        {
            status += "预约成功";
        }
        else if (of.m_orderData[i]["status"] == "-1")
        {
            status += "审核未通过, 预约失败";
        }
        else
        {
            status += "预约已取消";
        }
        cout << status << endl;
    }
    system("pause");
    system("cls");
}

```

9.2.2 测试功能

运行测试教师身份的查看所有预约功能

测试效果如图：



9.3 审核预约

9.3.1 审核功能实现

功能描述：教师审核学生的预约，依据实际情况审核预约

在Teacher.cpp中实现成员函数 void Teacher::validOrder()

代码如下：

```

//审核预约
void Teacher::validOrder()
{
    OrderFile of;
    if (of.m_Size == 0)
    {
        cout << "无预约记录" << endl;
        system("pause");
        system("cls");
        return;
    }
    cout << "待审核的预约记录如下：" << endl;

    vector<int>v;
    int index = 0;
    for (int i = 0; i < of.m_Size; i++)
    {
        if (of.m_orderData[i]["status"] == "1")
        {
            v.push_back(i);
            cout << ++index << "、 ";
            cout << "预约日期：周" << of.m_orderData[i]["date"];
            cout << " 时段：" << (of.m_orderData[i]["interval"] == "1" ? "上午" : "下午");
            cout << " 机房：" << of.m_orderData[i]["roomId"];
            string status = " 状态：" ; // 0取消的预约 1 审核中 2 已预约 -1 预约失败
            if (of.m_orderData[i]["status"] == "1")
            {
                status += "审核中";
            }
            cout << status << endl;
        }
    }
    cout << "请输入审核的预约记录,0代表返回" << endl;
    int select = 0;
    int ret = 0;
    while (true)
    {
        cin >> select;
        if (select >= 0 && select <= v.size())
        {
            if (select == 0)
            {
                break;
            }
            else
            {
                cout << "请输入审核结果" << endl;
                cout << "1、通过" << endl;
                cout << "2、不通过" << endl;
                cin >> ret;

                if (ret == 1)
                {
                    of.m_orderData[v[select - 1]]["status"] = "2";
                }
            }
        }
    }
}

```

```
        else
        {
            of.m_orderData[v[select - 1]]["status"] = "-1";
        }
        of.updateOrder();
        cout << "审核完毕! " << endl;
        break;
    }
}
cout << "输入有误, 请重新输入" << endl;
}

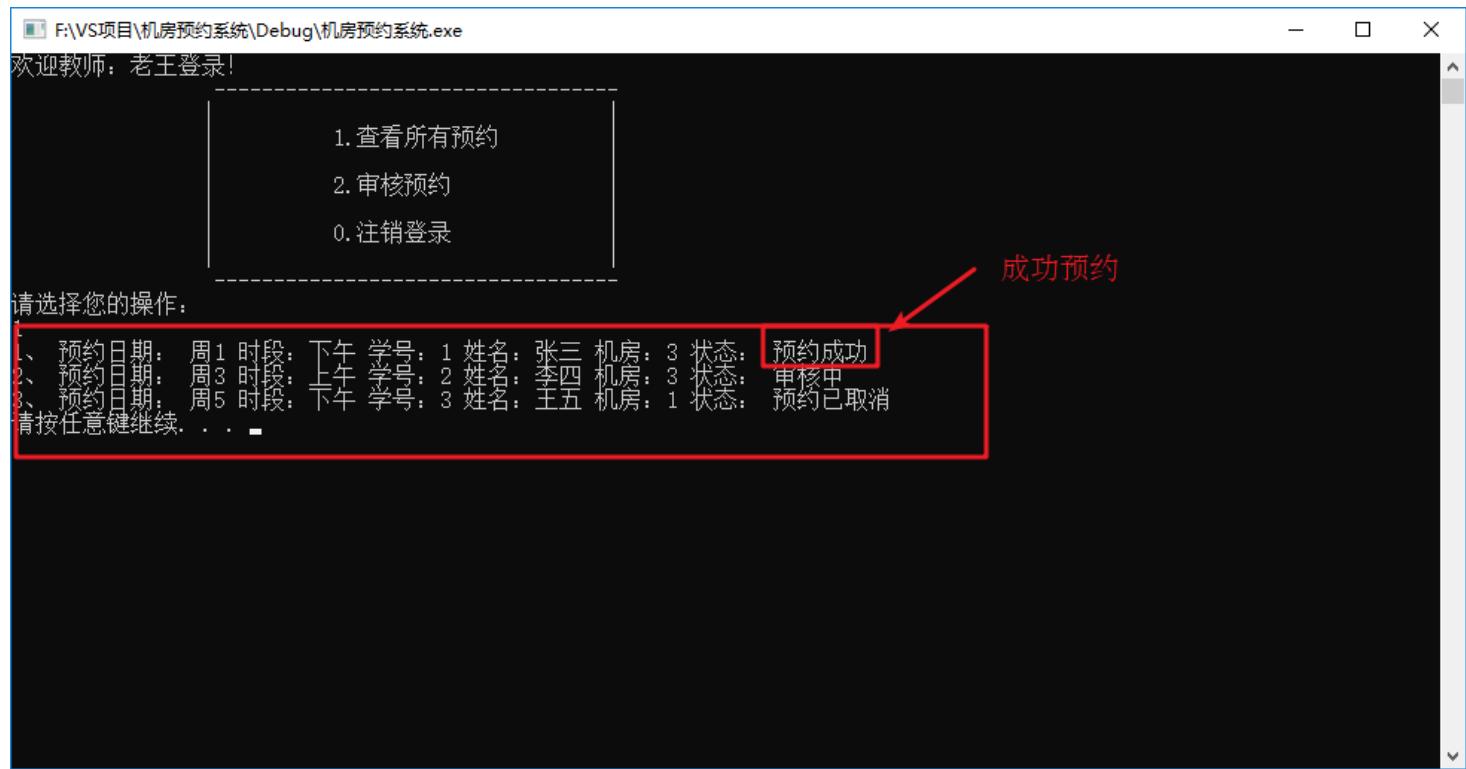
system("pause");
system("cls");
}
```

9.3.2 测试审核预约

测试 - 审核通过



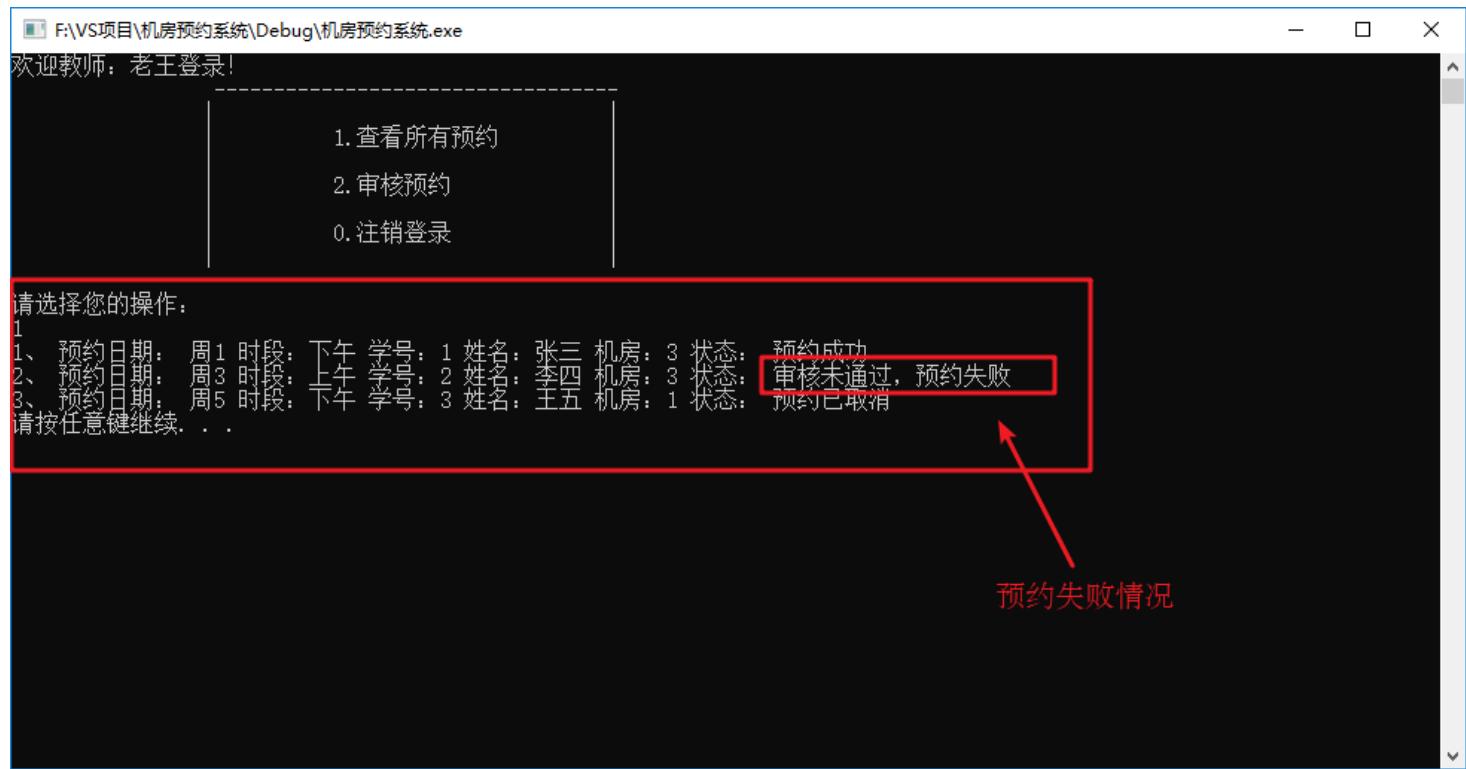
审核通过情况



测试-审核未通过



审核未通过情况：



学生身份下查看记录：



审核预约成功！

至此本案例制作完毕！ ^_^