目录

蒋贵良
标准 C++语言(11days)、QT 框架(7days)

《C++ Primer》
《C++程序设计原理与实践》

一 C++语言概述

1 历史背景

1）C++的江湖地位

java、C、python、C++

2）C++之父：Bjarne Stroustrup(1950--)

--> 1979，Cpre，为 C 语言增加了类的机制

--> 1983，C with Class(带类 C)，后来 C++

--> 1985，CFront 1.0，《The C++ Programming Language》

3）发展过程

--> 1987,GNU C++

--> 1990,Borland C++

--> 1992,Mircrosoft C++(VC/VS)

...

--> 1998,ISO C++98

--> 2003,ISO C++03

--> 2011,ISO C++11

--> 2014,ISO C++14

--> 2017,ISO C++17

*--> 2020,ISO C++20

2 应用领域

1）游戏开发

2）科学计算

3）网络和分布式应用

4）操作系统和设备驱动

5）其它...

3 C 和 C++

1）都是编译型语言

2）都是强类型语言，但是 C++更强

3）C++在很大程度兼容 C 语言，但是去除了一些不好的特性

4）C++增加了很多 C 语言中没有的好的特性，全面支持面向对象，比 C 语言更适合大型软件开发

二 第一个 C++程序

1 编译方式

1）gcc xx.cpp -lstdc++

2）g++ xx.cpp //推荐

01first.cpp

```
1 #include <iostream>
2 //#include <stdio.h>//C 风格
3 #include <cstdio>//C++风格
4 int main(void){
5       std::cout << "hello world!" << std::endl;
6       printf("hello world!\n");
```

```
7       return 0;
8 }
```

2 文件扩展名

1）.cpp //推荐

2）.cxx

3）.cc

4）.C

3 头文件

```
#include <iostream>
```
//和 I/O 相关操作都在该头文件中

注：在 C++开发中也可以使用标准 C 库的头文件，同时 C++里面提供一套不带.h 替换版本

eg:

```
#include <stdio.h>    <==> #include <cstdio>
#include <stdlib.h>  <==> #include <cstdlib>
#include <string.h>      <==> #include <cstring>
...
```

这种替换版本只能替换 C 库函数头文件,系统函数头文件不可替换

4 标准输出和输入

1）标准输出(cout)

```
int i = 123;
printf("%d\n",i);//C 语言
cout << i << endl;//C++
```

注：“<<”称为输出操作符

注：“endl”和“\n”类似，表示换行

```
-----------------------------------
int i=123,double d=4.56;
printf("%d,%lf\n",i,d);//C 语言
cout << i << ',' << d << endl;//C++
```

2）标准输入(cin)

```
int i = 0;
scanf("%d",&i);//C 语言
cin >> i;//C++
```

注：“>>”称为输入操作符

```
-----------------------------------
int i=0,double d=0.0;
scanf("%d%lf",&i,&d);//C 语言
cin >> i >> d;//C++
```

三 名字空间(命名空间)

1 名字空间的作用

1）避免名字冲突

2）划分逻辑单元

2 定义名字空间

```
namespace 空间名{
```

```
        名字空间成员 1;
        名字空间成员 2;
        ...
    }
```

注：名字空间成员可以全局变量、全局函数、类型、名字空间

注：“std”是标准 C++定义好的名字空间，称为标准名字空间，标准 C++库中所有的函数、变量、类型、对象都在其中。

3 名字空间成员的使用

1）作用域限定操作符 "::"

空间名::要访问的成员;

eg：

```
std::cout  //使用标准名字空间里面的 cout 成员
```

02namespace.cpp

```
#include <iostream>
namespace ns1{
    void func(void){
        std::cout << "ns1 的 func" << std::endl;
    }
}
namespace ns2{
    void func(void){
        std::cout << "ns2 的 func" << std::endl;
    }
}
int main(void){
    //func();//error,名字空间里面成员不能直接访问

    ns1::func();//ns1 的 func
    ns2::func();//ns2 的 func

    return 0;
}
```

2）名字空间指令

using namespace 空间名;

注：在该条指令以后的代码中，指定名字空间中的成员都可见，可以直接访问，省略"空间名::"

03namespace.cpp

```
#include <iostream>
using namespace std;//标准名字空间指令
namespace ns1{
    void func(void){
        cout << "ns1 的 func" << endl;
    }
```

```
 7 }
 8 namespace ns2{
 9         void func(void){
10                 cout << "ns2 的 func" << endl;
11         }
12 }
13 int main(void){
14         using namespace ns1;//名字空间指令
15         func();//ns1 的 func
16         using namespace ns2;
17         //func();//error,歧义错误(ambigous)
18         ns2::func();
19
20         return 0;
```

3）名字空间声明

using 空间名::要访问的成员;

注：将名字空间中特定的一个成员引入到声明所在的作用域，在该作用域中访问这个成员就如同访问自己作用域的成员一样，可以直接访问，省略"空间名::".

04namespace.cpp

```
 1 #include <iostream>
 2 using namespace std;//标准名字空间指令
 3 namespace ns1{
 4         void func(void){
 5                 cout << "ns1 的 func" << endl;
 6         }
 7 }
 8 namespace ns2{
 9         void func(void){
10                 cout << "ns2 的 func" << endl;
11         }
12 }
13 int main(void){
14         using namespace ns1;//名字空间指令
15         func();//ns1 的 func
16         using namespace ns2;
17         //func();//error,歧义错误
18         ns2::func();
19
20         return 0;
21 }
```

局部优先原则：名字空间声明比名字空间指令优先级高

4 全局作用域和无名名字空间 //了解

1）没有放在任何名字空间的成员，属于全局作用域，正常可以直接访问；但是如果和局部成员

名字一样，局部优先，这时如果还希望访问全局作用域的成员可以通过空"::"显式指明.
2）定义名字空间时可以没有名字，即为无名名字空间(匿名空间)，无名空间成员和全局作用域的成员一样，只是被限制在当前文件中使用。

05namepace.cpp

```
#include <iostream>
using namespace std;
namespace ns1{
	int num = 100;
}
namespace ns2{
	int num = 200;
}
//int num = 300;//全局作用域
namespace{
	int num = 300;//属于无名名字空间
}
int main(void){
	cout << num << endl;//300
	using ns1::num;//名字空间声明
	cout << num << endl;//100,局部优先
	cout << ns2::num << endl;//200
	cout << ::num << endl;//300 通过空"::"用于访问全局变量

	return 0;
}
```

5 名字空间的嵌套与合并 //了解
1）嵌套

```
namespace ns1{
	int num = 100;
	namespace ns2{
		int num = 200;
		namespace ns3{
			int num = 300;
		}
	}
}
cout << ns1::num << endl;//100
cout << ns1::ns2::num << endl;//200;
cout << ns1::ns2::ns3::num <<endl; //300
```

2）合并：定义名字空间时，如果和已存在空间名字一样，将会自动合并为同一个名字空间
eg:

```
namespace ns{//1.cpp
	void func1(void){..}
}
namespace ns{//2.cpp,自动和 1.cpp 的名字空间合并
```

```
        void func2(void){..}
    }
```

四 C++的结构体、联合体和枚举

1 C++的结构体

1）定义结构体变量可以省略 struct 关键字

2）在结构体中可以直接定义函数，称为成员函数(方法)，在成员函数中可以直接访问其它成员.

06struct.cpp

```
#include <iostream>
using namespace std;//标准名字空间指令
struct Teacher{
    //成员变量
    char name[20];
    int age;
    double salary;
    //成员函数
    void who(void){
        cout << "我叫" << name << ",今年" << age << "岁,工资为" <<
            salary << "元." << endl;
    }
};
int main(void){
    Teacher minwei={"闵卫",45,800.5};
    minwei.who();
    Teacher* pt = &minwei;
    pt->who();//(*pt).who()

    return 0;
}
```

2 C++的联合体//了解

1）定义联合体变量时可以省略 union 关键字

2）支持匿名联合(定义联合体的时候不取名字)

07union.cpp

```
#include <iostream>
using namespace std;
int main(void){
    union{//匿名联合
        unsigned int ui;
        unsigned char uc[4];
    };
    ui = 0x12345678;
    for(int i=0;i<4;i++){
        //hex:以十六进制方式打印整型数
        //showbase:显示十六进制标识"0x"
```

```
12              cout << hex << showbase << (int)uc[i] << ' ';
13      }
14      cout << endl;
15      //小端:内存高地址存放数据高位,低地址存放数据低位
16      //大端:内存高地址存放数据低位,低地址存放数据高位
17      return 0;
18  }
```

3 C++的枚举(提高代码的可读性)

1）定义枚举变量时可以省略 enum 关键字

2）C++中枚举是一种独立的数据类型，不能把枚举变量当做整型变量来使用。

```
                睡眠    运行    停止
    enum State{SLEEP=0,RUN=1,STOP=2};
    /*enum*/ State s;
    s = 2;//C:ok,C++:error
    s = STOP;//C:ok,C++:ok
```

O8enum.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main(void){
4      enum RGB{RED=10,GREEN,BLUE};
5      cout << RED << "," << GREEN << "," << BLUE << endl;//10,11,12
6      /*enum*/ RGB rgb;
7      //rgb = 12;//C:ok,C++:error
8      rgb = BLUE;//C:ok,C++:ok
9      cout << rgb << endl;//12
10     return 0;
11 }
```

五 字符串

1 回顾 C 语言的字符串

1）双引号字面值常量 "hello"

2）字符指针 char*

3）字符数组 char[]

Test.c

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void){
5      char* p1 = "hello";
6      char* p2 = "world";
7      //strcpy(p1,p2);//段错误,hello world 放在只读的代码区，无法访问
8
9      char a1[7] = "minwei";
10     char a2[12] = "youchengwei";
```

```
11
12      //strcpy(a1,a2);//越界访问,危险!数组在栈区
13
14      p1 = a1;//ok
15      a2 = p2;//error，数组名是一种指针常量，不能修改
16
17      return 0;
18  }
```

2 C++兼容 C 中字符串同时，增加 string 类型专门表示字符串

1）定义字符串

```
string s;//定义空字符串
string s = "xx";//定义同时初始化
--------------------------------
string s("xx");//和上面等价,了解
string s = string("xx");//和上面等价,了解
```

2）字符串拷贝：=

```
string s1 = "minwei";
string s2 = "youchengwei";
s1 = s2;
cout << s1 << endl;//"youchengwei"
```

3）字符串连接：+ +=

```
string s1 = "abc";
string s2 = "def";
string s3 = s1+s2;
cout << s3 << endl;//abcdef
```

4）字符串比较：== != > < >= <=

```
if(s1 == s2){...}
if(s1 != s2){...}
--------------------
string s1="minwei";//大
string s2="Youchengwei";
```

5）随机访问：[] //获取字符串中某个字符(char)

```
string str = "hello world";
str[0] = 'H';
str[6] = 'W';
cout << str << endl;//"Hello World"
```

6）常用的成员函数

```
size()/length() 获取字符串的长度(不包括'\0')
c_str() 获取字符串的起始地址(const char*),相当于将 string 转换为 C 风格 const char*

string str = "hello world";
cout << str.size() << endl;//11
cout << str.length() << endl;//11
-----------------
const char* cStr = str;//error
const char* cStr = str.c_str();
```

```
namespace std{
    struct string{
    成员函数
    char *str;
    };
}
```

练习：使用 string 表示字符串，从键盘读取字符串，统计里面包含字符'A'/'a'的个数?
提示：

```
    string  str;
    cin >> str;//从键盘读取字符串保存 str 中
```

09test.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int main(void){
 5     string str;
 6     //cin >> str;//碰到空白符(空格 tab 回车)结束
 7     getline(cin,str);//碰到'\n'结束
 8     int count = 0;
 9     for(int i=0;i<str.size();i++){
10         if(str[i]=='A' || str[i]=='a'){
11             ++count;
12         }
13     }
14     cout << "A(a)的个数:" << count << endl;
15     return 0;
16 }
```

练习：使用 string 表示字符串，从键盘读取字符串，实现字符串的翻转。
提示：

```
    输入：”abcdef”
    输出：”fedcba”
```

01string.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int main(void)
 5 {
 6     string str;
 7     cin >> str;
 8     int len = str.size();
 9     //abcdef->fedcba ==> 3
10     for(int i=0;i<len/2;i++){
11         str[i]          = str[i] ^ str[len-i-1];
```

```
12              str[len-i-1] = str[i] ^ str[len-i-1];
13              str[i]       = str[i] ^ str[len-i-1];
14      }
15      cout << str << endl;
16      return 0;
17 }
```

六 C++的布尔类型(bool)

1 bool 类型是 C++中的基本数据类型，专门表示逻辑值，逻辑真为 true，逻辑假为 false
2 bool 类型在内存占一个字节：1 表示 true，0 表示 false
3 bool 类型的变量可以接收任意类型表达式结果，其值非 0 则为 true，为 0 则为 false

```
bool func(){
        ...
        return true/false;//成功/失败
}
```

02bool.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void){
5       bool b = false;
6       cout << "b=" << b << endl;//0
7       cout << "size=" << sizeof(b) << endl;//1
8
9       b = 3+5;
10      cout << "b=" << b << endl;//1
11      b = 1.2*3.4;
12      cout << "b=" << b << endl;//1
13      int* p = NULL;//NULL->void*(0)
14      b = p;
15      cout << "b=" << b << endl;//0
16
17      return 0;
18 }
```

七 操作符别名 //了解

```
&&      <==>  and
||      <==>  or
^       <==>  xor
{       <==>  <%
}       <==>  %>
...
```

03operator.cpp

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

```
 2 using namespace std;
 3
 4 int main(void)<%
 5         int i=1;
 6         int j=0;
 7         if(i or j)<%
 8                 cout << "true" << endl;
 9         %>
10         else<%
11                 cout << "false" << endl;
12         %>
13         return 0;
14 %>
```

八 C++函数

1 函数重载

1）定义

在相同作用域，可以定义同名的函数，但是它们的参数表必须有所区分(类型和个数)，这样的多个函数将构成重载关系。

注:函数重载和返回类型无关。

04overload.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 void func(int i){
 5         cout << "func(int)" << endl;
 6 }
 7 void func(int i,int j){
 8         cout << "func(int,int)" << endl;
 9 }
10 void func(int a,float b){
11         cout << "func(int,float)" << endl;
12 }
13 int main(void){
14         func(10);
15         func(10,20);
16         func(10,1.23);//编译报错
17         func(10,1.23f);
18         //函数指针的类型决定匹配的重载版本
19         void (*pfunc)(int,float) = func;
20         pfunc(10,20);//func(int,float)
21
22         return 0;
23 }
```

05overload.cpp

```
#include <iostream>
using namespace std;
//char->int:升级转换
void bar(int i){
	cout << "bar(1)" << endl;
}
//char->const char:常量转换
void bar(const char c){
	cout << "bar(2)" << endl;
}
//short->char:降级转换
void fun(char c){
	cout << "fun(1)" << endl;
}
//short->int:升级转换
void fun(int i){
	cout << "fun(2)" << endl;
}
//省略号匹配
void hum(int i,...){
	cout << "hum(1)" << endl;
}
//double->int:降级转换
void hum(int i,int j){
	cout << "hum(2)" << endl;
}
int main(void){
	char c = 'A';
	bar(c);//bar(2)
	short s = 10;
	fun(s);//fun(2)

	int a=100;
	double d = 1.23;
	hum(a,d);//hum(2)
	return 0;
}
```

2）函数重载的匹配

调用重载关系的函数时，编译器会根据实参和形参的匹配程度，自动选择最优的重载版本，当前 g++编译器匹配的一般原则：

完全匹配>=常量转换>升级转换>降级转换>省略号匹配

完全匹配>常量转换：char* > const char*

完全匹配=常量转换：char = const char

升级转换过程中过分升级也会报错，比如：int -> long long int

3）函数重载的原理

C++的编译器会对函数进行换名，将参数表的类型信息整合到新的名字中，形成重载关系的函数参数表类型信息一定有所区别，解决函数重载和名字冲突的矛盾。

笔试题：C++中 extern "C"作用?

在 C++的函数前面加入 extern "C"声明，可以要求 C++编译器按照 C 语言的机制编译该函数，不对其进行换名，可以方便 C 程序直接调用该函数.

注：被 extern "C"声明的函数无法重载

cpp.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 extern "C" void hello(void){
5       cout << "Hello C++" << endl;
6 }
```

c.c

```
1 void hello(void);
2 //void _Z5hellov(void);
3
4 int main(void){
5       hello();
6       //_Z5hellov();
7       return 0;
8 }
```

```
/usr/lib/gcc/i686-linux-gnu/5/../../../i386-linux-gnu/crt1.o：在函数‘_start’中：
(.text+0x18)：对‘main’未定义的引用
collect2: error: ld returned 1 exit status
```

2 函数的哑元参数

1）定义

定义函数时，只有类型而没有变量名的形参称为哑元。

eg:

```
void func(int/*哑元*/){...}
```

2）使用哑元的场景

--》操作符重载，通过哑元参数区分前后++、-- //后面讲

--》为了兼容旧代码

算法库：

```
void math_func(int a,int b){}
```

使用者：

```
int main(){
    math_func(10,20);
    ...
    math_func(30,40);
    ...
```

```
        }
---------------------------------
算法库升级：
  void math_func(int a,int/*哑元*/){}
使用者：
        int main(){
            math_func(10,20);
            ...
            math_func(30,40);
            ...
        }
```

3 函数的缺省参数(默认实参)

1）可以为函数的部分或全部参数指定缺省值，调用该函数时，如果不给实参，就取缺省值作为默认实参。

```
    void func(int i = 0/*缺省参数*/){...}
```

08defArg.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 //函数声明
 4 void func(int a,int b=20,int c=30);
 5 //void func(int i){}//注意歧义错误
 6 int main(void){
 7     func(11,22,33);//11 22 33
 8     func(11,22);//11 22 30
 9     func(11);//11 20 30
10
11     return 0;
12 }
13 //函数定义
14 void func(int a,int b/*=20*/,int c/*=30*/){
15     cout << "a=" << a << ",b=" << b << ",c=" << c << endl;
16 }
```

2）靠右原则：如果函数的某个参数有缺省值，那么该参数右侧的所有参数都必须带有缺省值

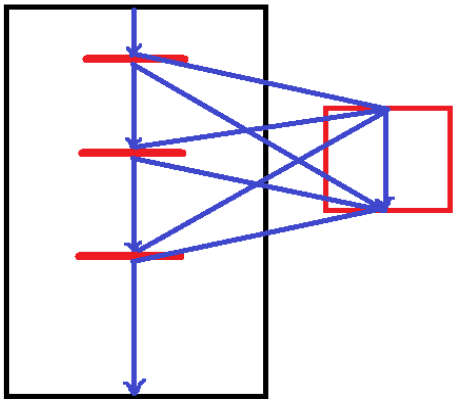
3）如果函数的定义和声明分开，缺省参数应该写在函数的声明部分，而定义部分不写。

4 内联函数(inline)

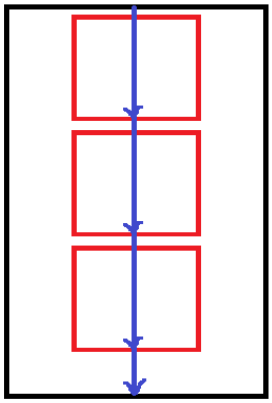
1）定义

使用 inline 关键字修饰的函数，表示这个函数是内联函数，编译器会尝试进行内联优化，可以避免函数调用开销，提高代码执行效率。

```
    inline void func(void){}//内联函数
```



正常函数调用



内联优化

2）使用说明
--》多次调用的小而简单的函数适合内联
--》调用次数极少或大而复杂的函数不适合内联
--》递归函数不适合内联
--》虚函数不适合内联//后面讲

注：内联只是一种建议而不是强制要求，一个函数能否内联优化主要取决于编译器，有些函数不加 inline 关键字也会默认处理为内联优化(struct 中的成员函数)，有些函数即便加了 inline 关键字，也会被编译器忽略(递归函数或虚函数)。

九 C++的动态内存管理

1 回顾 C 语言的动态内存管理(函数实现)
1）分配：malloc()
2）释放：free()

2 C++的动态内存管理(操作符实现)
1）分配：new/new[]
2）释放：delete/delete[]

09memory.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main(void){
5        //动态分配内存,保存一个 int
6        //int* pi = (int*)malloc(4);//C 语言
7        int* pi = new int;//C++
8        *pi = 100;
9        cout << *pi << endl;//100
10       //cout << "pi=" << pi << endl;
11       //free(pi);//C 语言
12       delete pi;//C++,防止内存泄露
13       pi = NULL;//避免使用野指针
```

```
14        //while(1);
15
16        //动态分配内存直接初始化
17        int* pi2 = new int(200);
18        cout << *pi2 << endl;//200
19        (*pi2)++;
20        cout << *pi2 << endl;//201
21        delete pi2;
22        pi2 = NULL;
23
24        return 0;
25  }
```

10memory.cpp

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main(void){
 5        //动态分配内存,保存 10 个整型数,返回的指针指向第一个元素(类似数组名)
 6        int* parr = new int[10];
 7        for(int i=0;i<10;i++){
 8              //*(parr+i) = i+1;
 9              parr[i] = i+1;
10              cout << parr[i] << ' ';
11        }
12        cout << endl;
13        delete[] parr;
14        parr = NULL;
15
16        //动态分配内存,保存 10 个整型数,并直接初始化(C++11)
17        int* parr2 = new int[10] {1,2,3,4,5,6,7,8,9,10};
18        for(int i=0;i<10;i++){
19              cout << parr2[i] << ' ';
20        }
21        cout << endl;
22        delete[] parr2;
23        parr2 = NULL;
24        return 0;
25  }
```

10memory.cpp:13:28: warning: extended initializer lists only available with -std=c++11 or -std=gnu++11

```
      int parr2 = new int[10]{1,2,3,4,5,6,7,8,9,10};
```

Shell:g++ 10memory.cpp -o 10 memory -std=gnu++11

11delete.cpp

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

```
2 using namespace std;
3
4 int main(void){
5       int* p1;
6       //delete p1;//delete 野指针,危险!
7
8       int* p2 = NULL;
9       delete p2;//delete 空指针,安全,但是没有意义
10
11      int* p3 = new int;
12      delete p3;
13      //delete p3;//重复 delete 同一地址,危险!(double free)
14
15      int* p4 = new int;
16      int* p5 = new int;
17      delete p4;
18      delete p5;
19      delete p4;//重复 delete 同一地址,虽然没有报错,但也要避免!
20
21      return 0;
22 }
```

十 C++的引用(Reference：参考，引用)

1 定义

1）引用即别名，引用就是某个变量的别名，对引用操作和对变量本身完全相同。

2）语法

类型 & 引用名 = 变量;

注：引用在定义时必须要初始化，初始化以后其绑定的目标变量不能再修改。

注：引用的类型和绑定的目标变量类型要一致

```
int a = 10;
int & b = a;//b 就是 a 的别名
++b;
cout << a << endl;//11

int c = 20;
b = c;//将 c 赋值给 b(a),而不是修改引用目标
```

12reference.cpp

```
1 #include <iostream>
2 using namespace std;
3 int main(void){
4       int a = 100;
5       int& b = a;//b 引用 a,b 就是 a 的别名
6       cout << "a=" << a << ",b=" << b << endl;
7       cout << "&a=" << &a << ",&b=" << &b << endl;
8       ++b;
```

```
 9       cout << "a=" << a << ",b=" << b << endl;//101,101
10       ++a;
11       cout << "a=" << a << ",b=" << b << endl;//102,102
12
13       //int& r;//error.引用定义时必须初始化
14       int c = 200;
15       b = c;//ok,但不是修改引用目标,仅是赋值操作
16       cout << "a=" << a << ",b=" << b << endl;//200
17       cout << "&a=" << &a << ",&b=" << &b << endl;
18
19       char& rc = c;//error,类型要一致
20
21       return 0;
22  }
```

2 常引用(万能引用)

1）定义引用时加 const 修饰，即为常引用，不能通过常引用修改引用的目标。

```
const 类型 & 引用名 = 变量;
类型 const & 引用名 = 变量;//和上面等价
eg：
int a = 10;
const int& b = a;//b 就是 a 的常引用
b++;//error
a++;//ok
```

2）普通的引用也可以称为左值引用，只能引用左值；而常引用也可以称为万能引用，即可以引用左值也可以引用右值。

注：关于左值和右值

左值(lvalue)：可以放在赋值运算符左侧，可以被取地址，可以被修改

右值(rvalue)：不能放在赋值运算符左侧，不可以被取地址，不可以被修改

13constRef.cpp

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main(void){
 5       //int& r1 = 100;//error,100 字面值常量是右值
 6       const int& r1 = 100;//ok
 7       cout << r1 << endl;//100
 8
 9       int i = 100;
10       //首先要 i 转换为 char 类型,转换结果保存到临时变量(右值)
11       //ri 实际引用转换结果的临时变量,而不是 i 本身
12       //char& ri = i;//error
13       const char& ri = i;//ok
14       cout << "&i=" << &i << endl;
15       cout << "&ri=" << (void*)&ri << endl;//默认情况下&ri 的类型是 void*,但是引
```

```
                //ri 前是 const  char  *  所以需要进行(void  *)转换，否则打印出来是字符串
16
17      return  0;
18  }
```

练习：验证下面表达式结果是左值还是右值?

```
int  a,b;
a+b;右值
a+=b;左值
-b;右值
++a;左值
a++;右值
------------------
int  func(void){return  ...}
func();//?右值
```

01lrvalue.cpp

```
1  #include  <iostream>
2  using  namespace  std;
3  int  func(void){
4       int  num  =  100;
5       return  num;//临时变量=num
6  }
7  int  main(void){
8       int  a=10,b=20;
9       //int&  r1  =  a+b;//error
10
11      int&  r2  =  (a+=b);//ok
12      //cout  <<  "&r2="  <<  &r2  <<  endl;
13      //cout  <<  "&a="  <<  &a  <<  endl;
14
15      //int&  r3  =  -b;//error
16
17      int&  r4  =  ++a;//ok
18
19      cout  <<  a++  <<  endl;//31
20      cout  <<  a  <<  endl;//32
21      //int&  r5  =  a++;//error
22
23      int&  r6  =  func();//error
24
25      return  0;
26  }
```

练习：区别(百度)? (笔试题)

1.指针和引用的定义和性质区别：

(1)指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟

原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。
(2)引用不可以为空，当被创建的时候，必须初始化，而指针可以是空值，可以在任何时候被初始化。
(3)可以有 const 指针，但是没有 const 引用；
(4)指针可以有多级，但是引用只能是一级（int **p；合法 而 int &&a 是不合法的）
(5)指针的值可以为空，但是引用的值不能为 NULL，并且引用在定义的时候必须初始化；
(6)指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。
(7)”sizeof 引用”得到的是所指向的变量(对象)的大小，而”sizeof 指针”得到的是指针本身的大小；
(8)指针和引用的自增(++)运算意义不一样；
(9)如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄漏；
2.指针和引用作为函数参数进行传递时的区别：
在讲引用作为函数参数进行传递时，实质上传递的是实参本身，即传递进来的不是实参的一个拷贝，因此对形参的修改其实是对实参的修改，所以在用引用进行参数传递时，不仅节约时间，而且可以节约空间。

3 引用型函数参数
1）可以将引用用于函数的形参，这时形参就是实参的别名，可以通过形参直接修改实参变量的值，同时还可以避免参数传值的开销，提高代码执行效率。
2) 引用型参数可能意外的修改实参的值，如果不希望修改实参本身，可以将形参定义为常引用，提高传参效率的同时还可以接收常量型的实参.

02refArg.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 void swap1(int* x,int* y){
 4       *x = *x ^ *y;
 5       *y = *x ^ *y;
 6       *x = *x ^ *y;
 7 }
 8 void swap2(int& x,int& y){
 9       x = x ^ y;
10       y = x ^ y;
11       x = x ^ y;
12 }
13 int main(void){
14       int a=3,b=5;
15       //swap1(&a,&b);
16       swap2(a,b);
17       cout << "a=" << a << ",b=" << b << endl;//a=5,b=3
18       return 0;
19 }
```

03refArg.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 struct Teacher{
```

```
    char name[100];
    int age;
    double salary;
};
void print(const Teacher& t){
    cout << t.name << "," << t.age/*++*/ << "," << t.salary << endl;
}
int main(void){
    const Teacher minwei={"闵卫",45,8000.5};//const 修饰是不可修改的左值，用法相当于右值
    print(minwei);
    print(minwei);
    return 0;
}
```

4 引用型函数返回值

1）可以将函数返回类型声明为引用，这时函数返回的结果就是 return 后面数据别名，可以避免函数返回值所带来的开销，提高代码执行效率

2）如果函数返回值是左值形式的引用，那么函数调用表达式结果就也是一个左值

注：不要从函数中返回局部变量的引用，因为所引用的内存会在函数返回以后被释放，可以从函数中返回成员变量、静态变量、全局变量的引用。

eg:

```
int func(void){
    ...
    return num;
}
func();//临时变量(右值)
func() = 123;//error
----------------------------
int& func(void){
    ...
    return num;
}
func();//num 自身(左值)
func() = 123;//ok
```

04refRet.cpp

```
#include <iostream>
using namespace std;
struct A{
    int num;
    int& func(void){
        return num;
    }
    int& func2(void){
        int data=100;
        return data;//危险!
```

```
11      }
12 };
13 int main(void){
14      A a = {100};
15      cout << a.num << endl;//100
16
17      //a.num = 200;
18      a.func() = 200;//ok
19      cout << a.num << endl;//200
20
21      //a.func2() = 200;
22
23      return 0;
24 }
```

5 引用和指针(笔试题)

1）从 C 语言角度看引用的本质，可以认为引用就是通过指针实现的，但是在 C++开发中不推荐使用指针而推荐使用引用。

```
int i = 100;
int* pi = &i;
int& ri = i;
*pi<==>ri
```

2）指针可以不做初始化，其目标可以在初始化以后进行改变(除了指针常量)；而引用定义时必须初始化，而且一旦初始化所引用目标不能再改变。

```
int a,b;
int* p;//ok
p = &a;//p 指向 a
p = &b;//p 指向 b
----------
int& r;//error
int& r = a;//r 引用 a
r = b;//仅是将 b 赋值给 r(a),而不是修改引用目标
```

//下面内容了解

3）可以定义指针的指针(二级指针)，但是不能定义引用的指针

4）可以定义指针的引用(指针变量的别名)，但是不能定义引用的引用

5）可以定义指针数组，但是不能定义引用数组，可以定义数组引用(数组的别名)

6）可以定义函数指针，也可以定义函数引用(函数的别名)，语法形式类似

05refPtr.cpp

```
1 #include <iostream>
2 using namespace std;
3 void func(int i){
4       cout << "func:" << i << endl;
5 }
6 int main(void){
7       int a = 123;
```

```
 8      int* p = &a;
 9      int** pp = &p;//指针的指针(二级指针)
10
11      int& r = a;
12      //int&* pr = &r;//error,引用的指针
13      int* pr = &r;//ok,仅是普通指针
14
15      int*& rp = p;//指针的引用
16      //int&& rr = r;//error,引用的引用
17      int& rr = r;//ok,仅是普通引用
18
19      int i=10,j=20,k=30;
20      int* parr[3] = {&i,&j,&k};//ok,指针数组
21      //int& rarr[3] = {i,j,k};//error,引用数组
22
23      int arr[3] = {i,j,k};
24      int(&rarr)[3] = arr;//ok,数组引用
25
26      void (*pfunc)(int) = func;//函数指针
27      void (&rfunc)(int) = func;//函数引用
28      pfunc(123);
29      rfunc(456);
30
31      return 0;
32  }
```

十一 类型转换

1 隐式类型转换

```
char c = 'A';
int i = c;//隐式类型
------------------------
void func(int i){}
func(c);//隐式类型
------------------------
int func(void){
        char c = 'A';
        return c;//隐式类型
}
```

2 显式类型转换

2.1 C++兼容 C 语言的强制转换

```
char c = 'A';
int i = (int)c;//强制转换,C 风格
int i = int(c);//强制转换,C++风格
```

06cast.cpp

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

```
2 using namespace std;
3
4 int main(void){
5       int* pi = NULL;
6       //char c = (long)pi;//C 风格,隐式转换无法实现，只能用显示转换
7       char c = long(pi);//C++风格
8       return 0;
9 }
```

2.2 C++增加了四种操作符形式类型转换，可以替换强制转换

1）静态类型转换：static_cast

语法：

目标类型变量 = static_cast<目标类型>(源类型变量)；

适用场景：

主要用于将 void*转换为其它类型的指针。

07static_cast.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void){
5       int num = 100;
6       void* pv = &num;
7       int* pi = static_cast<int*>(pv);//静态类型转换(合理)
8       cout << *pi << endl;//100
9
10      //double* pd = static_cast<double*>(pi);//静态类型转换(不合理)
11      //double* pd = (double*)pi;//静态类型转换(不合理)
12
13      return 0;
14 }
```

2）动态类型转换：dynamic_cast //后面讲

语法：

目标类型变量 = dynamic_cast<目标类型>(源类型变量)；

3）去常类型转换：const_cast

语法：

目标类型变量 = const_cast<目标类型>(源类型变量)；

适用场景：

主要用于去掉一个指针或引用的常属性.

08const_cast.cpp

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

```
int main(void){
    /* volatile 是标准 C 中关键字,表示易变的,告诉编译编译器,每次在使用被
     * volatile 修饰的变量时,需要从内存中重新读取,而不要直接使用寄存器
     * 中的副本,防止编译器优化引发的错误结果.*/
    volatile const int i = 100;
    int* pi = const_cast<int*>(&i);
    *pi = 200;
    cout << "i=" << i << ",*pi=" << *pi << endl;//200,200
    cout << "&i=" << (void*)&i << ",pi=" << pi << endl;

    return 0;
}
```

4）重解释类型转换：reinterpret_cast

语法：

目标类型变量 = reinterpret_cast<目标类型>(源类型变量);

适用场景：

--》在任意类型的指针和引用之间的显式转换

--》在指针和整型数之间进行转换

eg:已知物理地址 0x12345678,向该地址存放数据 100?

```
int* paddr = reinterpret_cast<int*>(0x12345678);
*paddr = 100;
```

09reinterpret_cast.cpp

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

int main(void){
    //"\000"->'\0'
    char buf[] = "0001\00012345678\000123456";
    struct HTTP{
        char type[5];
        char id[9];
        char passwd[7];
    };
    HTTP* ph = reinterpret_cast<HTTP*>(buf);
    cout << ph->type << endl;
    cout << ph->id << endl;
    cout << ph->passwd << endl;
    return 0;
}
```

小结:

1）慎用宏，可以使用 const、enum、inline 替换

#define PAI 3.14 ==》 const double PAI = 3.14;

```
#define STATE_SLEEP     0 \
```

```
#define  STATE_RUN        1   | ==> enum  STATE{SLEEP,RUN,STOP};
#define  STATE_STOP             2 /

#define  Max(a,b)  ((a)>(b)?(a):(b))
          ==> inline  int  Max(int  a,int  b){
                    return  a  >  b  ?  a  :  b;
                }
```

2）变量随用随声明同时初始化
3）尽量使用 new/delete 替换 malloc/free
4）少用 void*、指针计算、联合体、强制转换
5）尽量使用 string 表示字符串，少用 C 风格的字符串 char*/char[]

十二 类和对象 //了解

1 什么是对象

万物皆对象，任何一种事物都可以看做是对象.

2 如何描述对象

通过对象的属性(名词、数量词、形容词)和行为(动词)来描述对象。

3 面向对象程序设计

对自然世界中对象的观察引入到编程实践中的一种理念和方法，这种方法被称为"数据抽象",即在描述对象时把细节东西剥离出去，只考虑一般性的、有规律性的、统一性的东西。

4 什么是类

类是将多个对象共性提取出来定义的一种新的数据类型,是对对象属性和行为的抽象描述.

总结：

现实世界　　类　　虚拟世界

具体对象--抽象-->属性和行为--实例化-->具体对象

十三 类的定义和实例化

1 类的一般形式

```
struct/class  类名:继承方式  基类,...{
访问控制限制符:
    类名(形参表):初始化表{}    //构造函数(类中特殊的成员函数)
    ~类名(void){}                    //析构函数(类中特殊的成员函数)
    返回类型  函数名(形参表){}       //成员函数
    数据类型  变量名;                //成员变量
};
```

2 访问控制限制符，影响类中成员的访问位置

1）public:公有成员，任何位置都可以访问的成员
2）private:私有成员，只有类内部的成员函数才可以访问
3）protected:保护成员，当前类的内部和子类中可以访问 //后面讲

注：如果使用 struct 定义类，默认的访问控制属性是 public；如果使用 class 定义类，默认的访问控制属性是 private。(笔试题)

eg:

```
struct/class  XX{
    int  a;//默认
public:
    int  b;//公有成员
```

```
    private:
        int c;//私有成员
        int d;//私有成员
    public:
        int e;//公有成员
    private:
        int f;//私有成员
    };
```

10class.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 //原来定义结构体,现在定义类
 4 //struct Student{
 5 class Student{
 6 public:
 7     //成员函数:描述对象的行为
 8     void eat(const string& food){
 9         cout << "我在吃" << food << endl;
10     }
11     void sleep(int hour){
12         cout << "我睡了" << hour << "小时" << endl;
13     }
14     void learn(const string& course){
15         cout << "我在学" << course << endl;
16     }
17     void who(void){
18         cout << "我叫" << m_name << ",今年" << m_age << "岁,学号是" <<
19             m_no << endl;
20     }
21 public:
22     /* 类中的私有成员不能在外部直接访问,但是可以通过类似如下的公有成员函数
23      * 来间接访问,在函数体中可以对非法数据加以限定,控制业务逻辑的合理性.
24      * 这种编程思想就是"封装".*/
25     void setName(const string& newName){
26         if(newName == "二")
27             cout << "你才二" << endl;
28         else
29             m_name = newName;
30     }
31     void setAge(int newAge){
32         if(newAge < 0)
33             cout << "无效年龄" << endl;
34         else
35             m_age = newAge;
36     }
37     void setNo(int newNo){
```

```
38              if(newNo < 0)
39                  cout << "无效学号" << endl;
40              else
41                  m_no = newNo;
42          }
43  private:
44          //成员变量:描述对象的属性
45          string m_name;
46          int m_age;
47          int m_no;
48  };
49  int main(void){
50          Student s;//原来创建结构变量,现在称为创建对象/实例化对象/构造对象
51          /*s.m_name = "张三";
52          s.m_name = "二";
53          s.m_age = 25;
54          s.m_no = 10011;*/
55          s.setName("张三丰");
56          s.setName("二");
57          s.setAge(26);
58          s.setAge(-1);
59          s.setNo(10086);
60          s.setNo(-2);
61
62          s.who();
63          s.eat("兰州拉面");
64          s.sleep(8);
65          s.learn("C++编程");
66          return 0;
67  }
```

3 构造函数(Constructor)

1）语法

```
class 类名{
    类名(形参表){
        主要负责对象的初始化，即初始化成员变量;
    }
};
```

2）函数名和类名相同，没有返回类型

3）构造函数在创建对象时自动被调用，不能像普通的函数显式的调用

4）在每个对象的生命周期，构造函数一定会被调用，但也只会被调用一次

01constructor.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Student{
4  public:
```

```
    Student(const string& name,int age,int no){
        cout << "构造函数" << endl;
        m_name = name;
        m_age = age;
        m_no = no;
    }
    void who(void){
        cout << "我叫" << m_name << ",今年" << m_age << "岁,学号是"
            << m_no << endl;
    }
private:
    string m_name;
    int m_age;
    int m_no;
};
int main(void){
    //创建对象,这时将会自动调用 Student 的构造函数
    //(...):指明构造函数调用时需要的构造实参
    Student s("张三",26,10001);
    s.who();
    //构造函数不能通过对象显式的调用
    //s.Student("张三丰",27,10002);//error

    return 0;
}
```

练习：实现一个电子时钟类，要求使用构造函数接收当前系统时间初始化电子时钟的时间，并以秒为单位运行。

提示：

```
class Clock{
    public:
        Clock(time_t t){
            //将 time_t 的秒值转换为当前的系统时间并保存到 tm 结构体中
            tm* local = localtime(&t);
            时 = local->tm_hour;
            分 = local->tm_min;
            秒 = local->tm_sec;
        }
        void run(void){
            while(1){打印当前时间; 计时+1 秒; sleep(1);}
        }
    private:
        int 时,分,秒;
    };
    int main(void){
        //time(NULL)：获取当前的系统时间，返回 time_t 表示当前时间秒值(1970 到现在的秒值)
        Clock clock(time(NULL));
```

```
        clock.run();
    }
```

02clock.cpp

```
 1 #include <iostream>
 2 #include <cstdio>
 3 #include <ctime>
 4 #include <unistd.h>
 5 class Clock{
 6 public:
 7     Clock(time_t t){
 8         tm* local = localtime(&t);
 9         m_hour = local->tm_hour;
10         m_min = local->tm_min;
11         m_sec = local->tm_sec;
12     }
13     void run(void){
14         while(1){
15             printf("\r%02d:%02d:%02d",m_hour,m_min,m_sec);
16             fflush(stdout);//刷新标准输出缓冲区
17             if(60 == ++m_sec){
18                 m_sec = 0;
19                 if(60 == ++m_min){
20                     m_min = 0;
21                     if(24 == ++m_hour){
22                         m_hour = 0;
23                     }
24                 }
25             }
26             sleep(1);
27         }
28     }
29 private:
30     int m_hour;
31     int m_min;
32     int m_sec;
33 };
34 int main(void){
35     Clock clock(time(NULL));
36     clock.run();
37     return 0;
38 }
```

4 多文件编程：类的声明和定义分开

1）类的声明放在头文件中(.h)

2）类的实现放在源文件中(.cpp)

3）类的使用一般会在其它文件中

参考：project
Clock.h//类的声明
Clock.cpp//类的定义
main.cpp//类的使用

Clock.h

```
 1 #ifndef __CLOCK_H
 2 #define __CLOCK_H
 3
 4 #include <iostream>
 5 #include <cstdio>
 6 #include <ctime>
 7 #include <unistd.h>
 8 //类的声明:不包含函数体的实现
 9 class Clock{
10 public:
11     Clock(time_t t);
12     void run(void);
13 private:
14     int m_hour;
15     int m_min;
16     int m_sec;
17 };
18
19 #endif//__CLOCK_H
```

Clock.cpp

```
 1 #include "Clock.h"
 2 //类的实现:将类中的成员函数实现部分写在当前文件中
 3 //注:需要在函数名字前面加上"类名::",显式指明它们是属于类中的成员函数
 4 Clock::Clock(time_t t){
 5     tm* local = localtime(&t);
 6     m_hour = local->tm_hour;
 7     m_min = local->tm_min;
 8     m_sec = local->tm_sec;
 9 }
10 void Clock::run(void){
11     while(1){
12         printf("\r%02d:%02d:%02d",m_hour,m_min,m_sec);
13         fflush(stdout);//刷新标准输出缓冲区
14         if(60 == ++m_sec){
15             m_sec = 0;
16             if(60 == ++m_min){
17                 m_min = 0;
18                 if(24 == ++m_hour){
19                     m_hour = 0;
20                 }
```

```
21                  }
22              }
23              sleep(1);
24          }
25  }
```

main.cpp

```
1  #include "Clock.h"
2  //#include "Clock.h"
3
4  int main(void){
5          Clock clock(time(NULL));
6          clock.run();
7          return 0;
8  }
```

5 对象的创建和销毁

1）在栈区创建单个对象

类名 对象(构造实参表); //直接初始化

注：如果创建对象不需要构造实参，不要写空"()"

类名 对象 = 类名(构造实参表); //拷贝初始化，实际和上面等价

注：如果构造实参只有一个，可以简化为 类名 对象 = "构造实参"

2）在栈区创建多个对象(对象数组)

类名 对象数组[元素个数] = {类名(构造实参表),...};

3）在堆区创建单个对象

创建：类名* 对象指针 = new 类名(构造实参表);

注：使用 new 操作会为对象在堆区分配内存，然后自动的调用构造函数完成对象初始化；而如果是 malloc 只能分配，不会调用构造函数，不具备创建对象的能力。(笔试题)

销毁：delete 对象指针;

4）在堆区创建多个对象(对象数组)

创建：类名* 对象指针 = new 类名[元素个数]{类名(构造实参表),...};

销毁：delete[] 对象指针;

03object.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Student{
4  public:
5          Student(const string& name,int age,int no){
6                  cout << "构造函数" << endl;
7                  m_name = name;
8                  m_age = age;
9                  m_no = no;
```

```
10      }
11      void who(void){
12          cout << "我叫" << m_name << ",今年" << m_age << "岁,学号是"
13              << m_no << endl;
14      }
15  private:
16      string m_name;
17      int m_age;
18      int m_no;
19  };
20  int main(void){
21      //在栈区创建单个对象
22      Student s("张三",26,10001);
23      s.who();
24      Student s2 = Student("李四",25,10002);
25      s2.who();
26      //在栈区创建对象数组
27      Student sarr[3] = {
28          Student("张飞",27,10003),
29          Student("赵云",24,10004),
30          Student("马超",26,10005)};
31      sarr[0].who();
32      sarr[1].who();
33      sarr[2].who();
34      //在堆区创建单个对象
35      Student* ps = new Student("貂蝉",22,10006);
36      ps->who();//(*ps).who()
37      delete ps;
38      ps = NULL;
39      //在堆区创建多个对象
40      Student* parr = new Student[3]{
41          Student("小乔",23,10007),
42          Student("大乔",25,10008),
43          Student("孙尚香",20,10009)};
44      parr[0].who();//*(parr+0)->who()
45      parr[1].who();//*(parr+1)->who()
46      parr[2].who();//*(parr+2)->who()
47      delete[] parr;
48      parr = NULL;
49
50      return 0;
51  }
52
```

十四 构造函数和初始化列表

1 构造函数可以重载，也可以带有缺省参数

```
string s;//匹配无参构造函数
```

string s("hello");//匹配有参(const char*)构造函数

标准 C++库帮助手册网站：

http://www.cplusplus.com/reference/

04constructor.cpp

```
#include <iostream>
using namespace std;
class A{
public:
	/*A(void){
		cout << "A(void)" << endl;
		m_i = 0;
	}*/
	A(int i=0){
		cout << "A(int=0)" << endl;
		m_i = i;
	}
	int m_i;
};
int main(void){
	A a1;//匹配 A(void)
	cout << a1.m_i << endl;//0
	A a2(123);//匹配 A(int)
	cout << a2.m_i << endl;//123
	return 0;
}
```

2 缺省构造函数(无参构造函数)

1）如果类中没有定义任何构造函数，编译器会为该类提供一个缺省的无参构造函数：

--》对于基本类型的成员变量不做初始化

--》对于类类型的成员变量(成员子对象)，将会自动调用相应类的无参构造函数来初始化。

2）如果自己定义了构造函数，无论是否有参数，编译器都不会再提供无参构造函数。

05defCon.cpp

```
#include <iostream>
using namespace std;
class A{
public:
	A(void){
		cout << "A(void)" << endl;
		m_i = 0;
	}
	int m_i;
};
```

```
11 class B{
12 public:
13         //B(int j){}
14         int m_j;//基本类型的成员变量
15         A m_a;//类类型的成员变量(成员子对象)
16 };
17 int main(void){
18         B b;//匹配 B 中的缺省构造函数
19         cout << b.m_j << endl;//未初始化的结果
20         cout << b.m_a.m_i << endl;//0
21         return 0;
22 }
```

3 类型转换构造函数(单参构造函数)

```
class 类名{
        [explicit] 类名(源类型){...}
};
```

可以实现将源类型变量到当前类类型对象的转换。

注：可以使用 explicit 关键字修饰类型转换构造函数，可以强制要求使用它进行类型转换必须要显式的完成。

06castCons.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 class Integer{
5 public:
6         Integer(void){
7                 cout << "Integer(void)" << endl;
8                 m_i = 0;
9         }
10        //int->Integer:类型转换构造函数
11        /*explicit*/ Integer(int i){
12                cout << "Integer(int)" << endl;
13                m_i = i;
14        }
15        void print(void){
16                cout << m_i << endl;
17        }
18 private:
19        int m_i;
20 };
21 int main(void){
22        Integer i;
23        i.print();//0
24        //1)使用 Integer 类型转换构造函数,将 123 转换为 Integer 临时对象
25        //2)再使用临时对象对 i 进行赋值
```

```
26      i = 123;
27      i.print();//123
28
29      //上面隐式的类型转换代码可读性差,不推荐使用
30      //实际开发中推荐使用下面形式的显式类型转换
31      //i = (Integer)321;//C 风格
32      i = Integer(321);//C++风格(推荐)
33      i.print();
34
35      return 0;
36  }
```

4 拷贝(复制)构造函数

1）用一个已存在的对象构造同类型的新对象，这时会调用该类的拷贝构造函数

```
class 类名{
    类名(const 类名& that){...}
};
A a1(...);
A a2(a1);//匹配拷贝构造函数
A a2 = a1;//和上面完全等价
```

07cpCons.cpp

```
1  #include <iostream>
2  using namespace std;
3  class A{
4  public:
5      A(int i=0){
6          cout << "A(int=0)" << endl;
7          m_i = i;
8      }
9      A(const A& that){
10         cout << "A(const A&)" << endl;
11         m_i = that.m_i;
12     }
13     int m_i;
14 };
15 int main(void){
16     A a1(123);
17     A a2(a1);//拷贝构造
18     cout << a1.m_i << endl;//123
19     cout << a2.m_i << endl;//123
20     return 0;
21 }
```

2）如果一个类没有定义拷贝构造函数，那么编译器会为该类提供一个缺省的拷贝构造函数：

--》对于基本类型的成员变量，按字节复制

--》对于类类型的成员变量(成员子对象)，将会自动调用相应类的拷贝构造函数来初始化。

注：大多数情况下，并不需要自己定义拷贝构造函数，因为编译器缺省提供的已经很好用了

08cpCons.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class A{
 4 public:
 5       A(int i=0){
 6             cout << "A(int=0)" << endl;
 7             m_i = i;
 8       }
 9       A(const A& that){
10             cout << "A(const A&)" << endl;
11             m_i = that.m_i;
12       }
13       int m_i;
14 };
15 class B{
16 public:
17       A m_a;//成员子对象
18 };
19 int main(void){
20       B b1;//匹配 B 的无参构造函数
21       B b2 = b1;//匹配 B 的拷贝构造函数
22       cout << b1.m_a.m_i << endl;//0
23       cout << b2.m_a.m_i << endl;//0
24
25       return 0;
26 }
```

3）拷贝构造函数的调用时机
--》用已存在的对象作为同类型对象的构造实参
--》以对象形式向函数传递参数
--》从函数中返回对象

09cpCons.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class A{
 4 public:
 5       A(void){ cout << "A 的无参构造" << endl; }
 6       A(const A& that){ cout << "A 的拷贝构造" << endl; }
 7 };
 8 void foo(A a){
 9 }
10 A bar(void){
11       A a;//4 无参
```

```
12      cout << "&a=" << &a << endl;
13      return a;//拷贝(临时对象)
14  }
15  int main(void){
16      A a1;//1 无参
17      A a2 = a1;//2 拷贝
18      foo(a1);//3 拷贝
19      /* 正常情况,bar()返回 a 拷贝给临时对象,临时对象在拷贝给 a3,发生两次拷贝.
20       * 但是因为编译器优化,让 a3 直接引用 bar 返回的 a,不再发生拷贝.
21       * g++ 09cpCons.cpp -fno-elide-constructors */去掉优化
22      A a3 = bar();//拷贝
23      cout << "&a3=" << &a3 << endl;
24      return 0;
25  }
26  //一共会调用多少次构造函数?
27  //A 3 次, B 4 次, C 5 次, D 6 次(正确)
```

5 初始化列表

1）语法

```
class 类名{
    类名(形参表):成员变量 1(初值),成员变量 2(初值),...{}
};
```

eg：

```
class Student{
public:
    //先定义成员变量，再赋初值
    Student(const string& name,int age,int no){
        m_name = name;
        m_age = age;
        m_no = no;
    }
    //定义成员变量同时初始化
    Student(const string& name,int age,int no):
        m_name(name),m_age(age),m_no(no){}
private:
    string m_name;
    int m_age;
    int m_no;
};
```

2）需要使用初始化列表的场景

--》如果类中包含了类类型的成员变量(成员子对象)，并希望以有参的方式对其初始化，则必须通过初始化列表来初始化该成员变量.

--》如果类中包含了“const”或“引用”型的成员变量，必须在初始化列表中显式的初始化

10initlist.cpp

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

```
 3 class A{
 4 public:
 5       A(int i){
 6             cout << "A 的构造函数" << endl;
 7             m_i = i;
 8       }
 9       int m_i;
10 };
11 class B{
12 public:
13       //首先根据初始化列表完成成员子对象的创建
14       //再执行 B 自己的构造函数代码
15       B(void):m_a(123){
16             cout << "B 的构造函数" << endl;
17       }
18       A m_a;//成员子对象
19 };
20 int main(void){
21       B b;
22       cout << b.m_a.m_i << endl;
23       return 0;
24 }
```

11initlist.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 int num = 200;
 4 class A{
 5 public:
 6       /*A(void){
 7             ci = 100;
 8             ri = num;
 9       }*/
10       A(void):ci(100),ri(num){}
11       const int ci;
12       int& ri;
13 };
14 int main(void){
15       A a;
16       cout << a.ci << "," << a.ri << endl;
17       return 0;
18 }
```

笔试题：

01initlist.cpp

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

```
#include <cstring>
using namespace std;
class Dummy{
public:
    Dummy(const char* str)
        //:m_str(str),m_len(m_str.size()){}
        //:m_str(str),m_len(strlen(str)){}
        :m_str(str?str:""),m_len(strlen(str?str:"")){}<--------+防止传递空指针
    size_t m_len;                                              |
    string m_str;                                             |
};                                                             |
int main(void){                                               |
    Dummy d("minwei");                                         |
    //Dummy d(NULL);-----------------------------------------+
    cout << d.m_str << "," << d.m_len << endl;//minwei,6
    return 0;
}
```

注：成员变量的初始化顺序由声明顺序决定，而与初始化列表的顺序无关，所以不要使用一个成员变量去初始化另一个成员变量.

十五 this 指针和常成员函数

1 this 指针

1) this 是一个关键字，其本质就是一个指针形式的形参变量。在类中的成员函数(包含构造函数、析构函数)中都会隐藏一个该类类型的指针参数，即为 this.

2) 对于普通的成员函数，this 指针就指向该成员函数的调用对象；对于构造函数，this 指针就是指向正在创建的对象.

eg：

```
class A{
public:
    //void func(A* this)
    void func(void){
        //cout << this->m_data << endl;
        cout << m_data << endl;
    }
    int m_data;
};
int main(void){
    A a(123);
    a.func();//A::func(&a)
}
```

02this.cpp

```
#include <iostream>
using namespace std;
class Teacher{
public:
```

```
    Teacher(const string& name,int age,double salary)
        :m_name(name),m_age(age),m_salary(salary){
        cout << "构造函数:" << this << endl;
    }
    void print(void){
        cout << m_name << "," << m_age << "," << m_salary << endl;
        //this 指针可以在成员函数里面显式使用,和上面等价
        cout << this->m_name << "," << this->m_age << "," <<
            this->m_salary << endl;
    }/*print 在编译器处理后,会变成类似如下:
    void print(Teacher* this){
        cout << this->m_name << "," << this->m_age << "," <<
            this->m_salary << endl;
    }*/
private:
    string m_name;
    int m_age;
    double m_salary;
};
int main(void){
    Teacher t1("游成伟",35,3000.5);
    Teacher t2("闵卫",45,4000.5);
    t1.print();//Teacher::print(&t1);不可自己调用
    t2.print();//Teacher::print(&t2);不可自己调用
    cout << "&t1=" << &t1 << endl;
    cout << "&t2=" << &t2 << endl;
    return 0;
}
```

3）必须显式使用 this 指针的场景
--》区分作用域
--》从成员函数中返回调用对象自身（返回自引用） //重点掌握
--》在类的内部销毁(堆)对象自身（对象自销毁）
--》作为函数实参，实现不同对象之间的交互 //了解

03this.cpp

```
#include <iostream>
using namespace std;
class A{
public:
    A(void):data(0){}
    //通过 this 区分作用域:如果函数的参数变量名和成员变量名相同,在成员函数
    //里面将会优先访问参数变量,这时如果希望访问成员的话,需要显式使用 this
    void set(int data){
        this->data = data;
    }
    void print(void){
```

```
12              cout << data << endl;
13      }
14  private:
15      int data;
16  };
17  int main(void){
18      A a;
19      a.print();//0
20      a.set(123);
21      a.print();//123;
22      return 0;
23  }
```

04this.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  class Counter{
5  public:
6      Counter(int count=0):m_count(count){}
7      Counter& add(void){//Counter& add(Counter* this)
8          ++m_count;
9          //this 指向成员函数的调用对象,*this 就是调用对象自身
10         return *this;//返回自引用
11     }
12     void print(void){
13         cout << "计数值:" << m_count << endl;
14     }
15     //对象自销毁函数
16     void destroy(void){
17         cout << "this=" << this << endl;
18         delete this;
19     }
20 private:
21     int m_count;
22 };
23 int main(void){
24     Counter cn;
25     //Counter::add(&cn)
26     cn.add().add().add();
27     cn.print();//3
28
29     Counter* pcn = new Counter;
30     pcn->add();
31     pcn->print();//1
32     //delete pcn;
33     cout << "pcn=" << pcn << endl;
```

```
34        pcn->destroy();
35
36        return 0;
37 }
```

05this.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class Student;//短视声明
 5 class Teacher{
 6 public:
 7        void educate(Student* stu);
 8        void reply(const string& answer);
 9 private:
10        string m_answer;
11 };
12 class Student{
13 public:
14        void ask(const string& ques,Teacher* teach);
15 };
16 void Teacher::educate(Student* stu){
17        //通过 this 指针,将教师对象的地址传给学生类的 ask 函数
18        stu->ask("什么是 this 指针?",this);//(1)
19        cout << "学生回答:" << m_answer << endl;//(5)
20 }
21 void Teacher::reply(const string& answer){
22        m_answer=answer;//(4)
23 }
24 void Student::ask(const string& ques,Teacher* teach){
25        cout << "问题:" << ques << endl;//(2)
26        teach->reply("this 就是指向该成员函数的调用对象");//(3)
27 }
28 int main(void){
29        Teacher t;
30        Student s;
31        t.educate(&s);
32        return 0;
33 }
```

2 常成员函数

1）在一个普通的成员函数的参数表后面加 const 修饰，这个成员函数就是常成员函数(常函数)

```
class 类名{
        返回类型 函数名(参数表) const {函数体}
};
```

2）常成员函数中的 this 指针是一个常指针，不能在常成员函数中修改成员变量的值。

注：被 mutable 关键字修饰的成员变量可以常函数中修改

06constfun.cpp

```
#include <iostream>
using namespace std;
class A{
public:
    A(int data=0):m_data(data){}
    //void print(const A* this)
    void print(void)const{//常成员函数
        //cout << this->m_data << endl;
        cout << m_data/*++*/ << endl;

        //可以通过 const_cast 去掉 this 的常属性,修改成员变量
        //const_cast<A*>(this)->m_spec = 200;

        //被 mutable 修饰的成员变量,也可以直接修改
        m_spec = 200;                                    ^
        cout << m_spec << endl;                          |
    }                                                    |
private:                                                 |
    int m_data;                                          |
    mutable int m_spec;   ----------------------+
};
int main(void){
    A a(100);
    a.print();//100
    a.print();//101
    return 0;
}
```

3）非 常对象既可以调用常函数也可以调用非 常函数，但是常对象只能调用常函数，不能调用非常函数。//重点掌握

注：常对象也包括常指针或常引用

07constfunc.cpp

```
#include <iostream>
using namespace std;
class A{
public:
    //void func1(const A* this)
    void func1(void) const {
        cout << "常函数" << endl;
        //func2();//this->func2(),error
    }
    //void func2(A* this)
    void func2(void) {
        cout << "非常函数" << endl;
```

```
13      }
14 };
15 int main(void){
16      A a;
17      a.func1();//A::func1(&a),A*
18      a.func2();//A::func2(&a),A*
19      const A ca = a;
20      ca.func1();//A::func1(&ca),const A*
21      //ca.func2();//A::func2(&ca),const A*,error
22      const A* pa = &a;//pa 常指针
23      pa->func1();
24      //pa->func2();//error
25      const A& ra = a;//ra 常引用
26      ra.func1();
27      //ra.func2();//error
28
29      return 0;
30 }
```

4）同一个类中，函数名和形参表相同的成员函数，其常版本和非常版本可以构成有效的重载关系，常对象匹配常版本，非常对象匹配非常版本。

08constFunc.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class A{
 4 public:
 5      void func(void) const {
 6           cout << "func 的常版本" << endl;
 7      }
 8      void func(void) {
 9           cout << "func 的非常版本" << endl;
10      }
11 };
12 int main(void){
13      A a;
14      a.func();//非常版本
15      const A ca = a;
16      ca.func();//常版本
17      return 0;
18 }
```

十六 析构函数(destructor)

1 语法

```
class 类名{
public:
     ~类名(void){
```

```
            //主要负责清理对象生命周期中的动态资源
        }
};
```

1）函数名必须是"~类名"
2）没有返回类型，也没有参数
3）不能被重载，一个类只能有一个析构函数

2 析构函数在对象销毁时，自动被调用执行
1）栈对象离开所在作用域时，被作用域终止右花括号"}"销毁

```
int func(void){
    A a;
    return 0;
}//负责销毁
int func(void){
    if(...){
        A a;
    }//负责销毁
    return 0;
}
```

2）堆对象在执行 delete 操作符，被 delete 销毁。

```
int main(void){
    A* pa = new A;
    ...
    delete pa;//负责销毁
    return 0;
}
```

注：delete 操作符会自动调用析构函数，再释放对象自身内存；而如果是 free()只能释放内存，不会调用析构，不具备销毁对象的能力。(笔试题)

09destructor.cpp

```
#include <iostream>
using namespace std;
class Integer{
public:
    Integer(int i = 0):m_pi(new int(i)){
        //m_pi = new int(i);
        cout << "动态分配资源" << endl;
    }
    void print(void) const {
        cout << *m_pi << endl;
    }
    ~Integer(void){//析构函数
        cout << "释放动态资源" << endl;
        delete m_pi;
    }
private:
    int* m_pi;
```

```
18 };
19 int main(void){
20         Integer i(100);
21         i.print();//100
22
23         Integer* pi = new Integer(200);
24         pi->print();
25         delete pi;//销毁堆对象,调用析构函数
26
27         return 0;
28 }//销毁栈对象,调用析构函数
```

3 缺省析构函数
1）如果类中自己没有定义析构函数，那么编译器会为该类提供一个缺省的析构函数
2）对于基本类型的成员变量，什么也不做；对于类类型的成员变量(成员子对象)，将会自动的调用相应类的析构函数来销毁该子对象.

4 对象创建和销毁的过程
1）创建
--》分配内存
--》构造成员子对象
--》执行构造函数代码
2）销毁
--》执行析构函数代码
--》析构成员子对象
--》释放内存

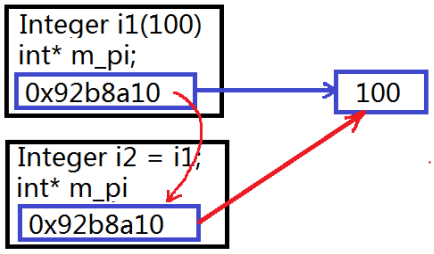
10destructor.cpp(笔试题)

```
 1 #include <iostream>
 2 using namespace std;
 3 class A{
 4 public:
 5         A(void){
 6                 cout << "A(void)" << endl;//(1)
 7         }
 8         ~A(void){
 9                 cout << "~A(void)" << endl;//(4)
10         }
11 };
12 class B{
13 public:
14         B(void){
15                 cout << "B(void)" << endl;//(2)
16         }
17         ~B(void){
18                 cout << "~B(void)" << endl;//(3)
19         }
```

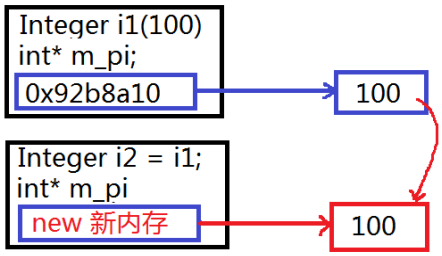
```
20      A m_a;//成员子对象
21  };
22  int main(void){
23      B b;
24      return 0;
25  }
```

十七 拷贝构造和拷贝赋值(笔试题)

1 深拷贝和浅拷贝



浅拷贝：只复制对象里面指针本身地址编号



深拷贝：复制对象里面指针所指向的数据

参考 copy.png

11copy.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Integer{
4  public:
5      Integer(int i = 0):m_pi(new int(i)){
6      }
7      void print(void) const {
8          cout << *m_pi << endl;
9      }
10     ~Integer(void){//析构函数
11         cout << "析构函数:" << m_pi << endl;
12         delete m_pi;
13     }
14     //缺省拷贝构造函数(浅拷贝)
15     /*Integer(const Integer& that){
16         cout << "缺省拷贝构造函数" << endl;
17         //i2.m_pi = i1.m_pi
18         m_pi = that.m_pi;
19     }*/
20     //自定义拷贝构造函数(深拷贝)
21     Integer(const Integer& that){
22         cout << "自定义拷贝构造函数" << endl;
23         m_pi = new int;
24         *m_pi = *that.m_pi;
```

```
25      }
26  private:
27      int*  m_pi;
28  };
29  int  main(void){
30      Integer  i1(100);
31      Integer  i2=i1;//拷贝构造
32      i1.print();//100
33      i2.print();//100
34      return  0;
35  }
```

1) 如果类中包含了指针形式的成员变量，缺省的拷贝构造只是复制了指针变量本身的地址编码，没有复制指针所指向的数据，这种拷贝方式被称为浅拷贝。
2）浅拷贝将会导致不同对象之间的数据共享，逻辑混乱，另外在对象销毁时，执行析构函数，还可能会引发"double free"异常，导致进程终止。
3）为了避免浅拷贝的问题，就必须自己定义一个支持复制指针所指向数据的拷贝构造函数，即深拷贝

练习：实现字符串(String)类,包括构造函数、析构函数、拷贝构造函数(笔试题)
提示：

```
class  String{
public:
    //构造函数
    String(const  char*  str=NULL){
        m_str  =  new  char[strlen(str)+1];
        strcpy(m_str,str);
    }
    //析构函数
    //拷贝构造
private:
    char*  m_str;
};
String  s  =  "minwei";
```

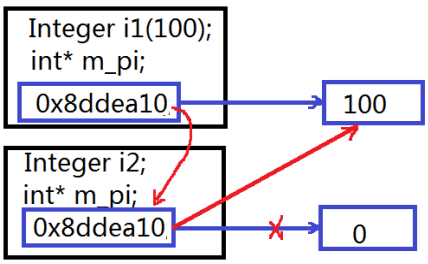
12string.cpp

```
1  #include  <iostream>
2  #include  <cstring>
3  using  namespace  std;
4
5  class  String{
6  public:
7      //构造函数
8      String(const  char*  str=NULL){
9          m_str  =  new  char[strlen(str?str:"")+1];
10         strcpy(m_str,str?str:"");
11     }
```

```
12      //析构函数
13      ~String(void){
14          if(m_str){
15              delete[] m_str;
16              m_str = NULL;
17          }
18      }
19      //拷贝构造
20      String(const String& that){
21          m_str = new char[strlen(that.m_str)+1];
22          strcpy(m_str,that.m_str);
23      }
24      //测试
25      void print(void)const{
26          cout << m_str << endl;
27      }
28  private:
29      char* m_str;
30  };
31  int main(void){
32      String s = "minwei";
33      s.print();
34      String s2 = s;//拷贝构造
35      s2.print();
36
37      return 0;
38  }
```

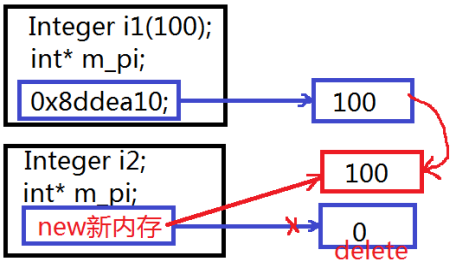
2 拷贝赋值

参考 copy2.png



i2 = i1;//i2.operator=(i1)

浅拷贝:数据共享，double free，内存泄漏



i2 = i1;//i2.operator=(i1)

深拷贝:释放就内存;分配新内存;拷贝新数据.

01copy.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Integer{
4  public:
```

```
5       Integer(int i = 0):m_pi(new int(i)){
6       }
7       void print(void) const {
8           cout << *m_pi << endl;
9       }
10      ~Integer(void){//析构函数
11          cout << "析构函数:" << m_pi << endl;
12          delete m_pi;
13      }
14      //缺省拷贝构造函数(浅拷贝)
15      /*Integer(const Integer& that){
16          cout << "缺省拷贝构造函数" << endl;
17          //i2.m_pi = i1.m_pi
18          m_pi = that.m_pi;
19      }*/
20      //自定义拷贝构造函数(深拷贝)
21      Integer(const Integer& that){
22          cout << "自定义拷贝构造函数" << endl;
23          m_pi = new int;
24          *m_pi = *that.m_pi;
25      }
26      //缺省拷贝赋值函数(浅拷贝)
27      //i2=i1 ==> i2.operator=(i1)
28      /*Integer& operator=(const Integer& that){
29          cout << "缺省拷贝赋值函数" << endl;
30          if(&that != this){//防止自赋值
31              m_pi = that.m_pi;
32          }
33          return *this;
34      }*/
35      //自定义拷贝赋值函数(深拷贝)
36      //i2=i1 ==> i2.operator=(i1)
37      Integer& operator=(const Integer& that){
38          cout << "自定义拷贝赋值函数" << endl;
39          if(&that != this){//防止自赋值
40              delete m_pi;//释放旧内存
41              m_pi = new int;//分配新内存
42              *m_pi = *that.m_pi;//拷贝新数据
43          }
44          return *this;//返回自引用
45      }
46  private:
47      int* m_pi;
48  };
49  int main(void){
50      Integer i1(100);
51      Integer i2;
```

```
52      i2.operator=(i1);
53      //i2 = i1;//拷贝赋值
54      i1.print();//100
55      i2.print();//100
56      return 0;
57  }
```

1）当两个对象进行赋值运算时，比如"i2=i1",编译器会将其处理为 i2.operator=(i1)成员函数调用形式，其中“operator=”被称为拷贝赋值函数，由该函数实现两个对象的赋值操作，其返回结果就是表达式的结果.
2）如果自己没有定义拷贝赋值函数，那么编译器会为该类提供一个缺省的拷贝赋值函数，但是缺省的实现是浅拷贝，只会复制指针本身的地址编号，不会复制指针所指向的数据，可能有"数据共享"，"double free"，"内存泄漏"的问题
3）为了避免缺省浅拷贝赋值函数的问题，必须自定义深拷贝赋值函数：

```
类名& operator=(const 类名& that){
    if(&that != this){//防止自赋值
        释放旧内存;
        分配新内存;
        拷贝新数据;
    }
    return* this;//返回自引用
}
```

练习：为 String 增加拷贝赋值函数(笔试题)---->考的多

02string.cpp

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class String{
6  public:
7      //构造函数
8      String(const char* str=NULL){
9          m_str = new char[strlen(str?str:"")+1];
10         strcpy(m_str,str?str:"");
11     }
12     //析构函数
13     ~String(void){
14         if(m_str){
15             delete[] m_str;
16             m_str = NULL;
17         }
18     }
19     //拷贝构造
20     String(const String& that){
21         m_str = new char[strlen(that.m_str)+1];
22         strcpy(m_str,that.m_str);
```

```
23      }
24      //拷贝赋值
25      //s2=s3 ==> s2.operator=(s3);
26      String& operator=(const String& that){
27          if(&that != this){
28              delete[] m_str;
29              m_str = new char[strlen(that.m_str)+1];
30              strcpy(m_str,that.m_str);
31              /*char* str = new char[strlen(that.m_str)+1];
32              delete[] m_str;
33              m_str = strcpy(str,that.m_str);*/
34              /*String tmp(that);
35              swap(m_str,tmp.m_str);*/
36          }
37          return *this;
38      }
39      //测试
40      void print(void)const{
41          cout << m_str << endl;
42      }
43  private:
44      char* m_str;
45  };
46  int main(void){
47      String s = "minwei";
48      s.print();
49      String s2 = s;//拷贝构造
50      s2.print();
51      String s3 = "今天是星期六!";
52      //s2.operator=(s3);
53      s2 = s3;//拷贝赋值,s2.operator=(s3)
54      s2.print();
55
56
57      return 0;
58  }
```

十八 静态成员(static)

1 静态成员变量

1）语法

```
class 类名{
    static 数据类型 变量名;//声明
};
数据类型 类名::变量名 = 初值;//定义和初始化
```

2）普通的成员变量属于对象，而静态成员变量不属于对象.

3）普通的成员变量在对象构造时定义和初始化，而静态成员变量需要在类的外部单独的定义和初始化

4）静态成员变量其本质和全局变量类似，被放在全局区，可以把静态成员变量理解为被限制在类中使用的全局变量。
5）使用方法：
类名::静态成员变量;//推荐
对象名.静态成员变量;//本质和上面等价

03static.cpp

```
#include <iostream>
using namespace std;
class A{
public:
	//普通成员变量由构造函数完成定义和初始化
	A(int data):m_data(data){}

	int m_data;//普通成员变量
	static int s_data;//静态成员变量
	static const int spec = 20;//特殊(了解)
};
//静态成员变量需要在类的外部单独的定义和初始化
int A::s_data = 20;

int main(void){
	A a1(10);
	//普通的成员变量属于对象,静态成员变量不属于对象
	cout << sizeof(a1) << endl;//4
	//普通的成员需要通过对象才能访问
	cout << a1.m_data << endl;//10
	//静态成员变量可以通过"类名::"直接访问
	cout << A::s_data << endl;//20
	//静态成员变量也可以通过对象访问
	cout << a1.s_data << endl;//ok

	A a2(10);
	a1.m_data = 100;
	a1.s_data = 200;
	cout << a2.m_data << endl;//10
	cout << a2.s_data << endl;//200
	return 0;
};
```

2 静态成员函数
1）语法

```
class 类名{
	static 返回类型 函数名(形参表){函数体}
};
```

2）静态成员函数没有 this 指针，不能有 const 属性修饰(报错)，可以把静态成员函数理解为被限制在类中使用的全局函数。

3）使用方法

类名::静态成员函数(实参表);//推荐

对象名.静态成员函数(实参表);//本质和上面等价

注：静态成员函数只能访问静态成员，不能访问非静态成员；普通成员函数既可以访问静态成员也可以访问非静态成员。(静态成员函数没有 this 指针)

04static.cpp

```
#include <iostream>
using namespace std;
class A{
public:
    A(void):m_data(10){}
    static void func1(void){
        cout << "静态成员函数" << endl;
        cout << s_data << endl;
        //cout << m_data << endl;//error
    }
    void func2(void){
        cout << "非静态成员函数" << endl;
        cout << s_data << endl;
        cout << m_data << endl;
    }
    static int s_data;
    int m_data;
};
int A::s_data = 20;

int main(void){
    A::func1();
    A a;
    a.func2();
    return 0;
}
```

3 单例模式

1）概念

一个类只允许存在唯一的对象，并提供该对象的访问方法.

2）实现思路

--》禁止在类的外部创建对象：私有化构造函数

--》类的内部维护唯一的对象：静态成员变量

--》提供单例对象的访问方法：静态成员函数

3）创建方式

--》饿汉式：单例对象无论用或不用，在程序启动即创建

05hungry.cpp

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

```
3  //单例模式:饿汉式
4  class Singleton{
5  public:
6          void print(void)const{
7                  cout << m_data << endl;
8          }
9  private:
10         //1)私有化构造函数(包括拷贝构造函数)
11         Singleton(int data=0):m_data(data){
12                 cout << "单例对象被创建了" << endl;
13         }
14         Singleton(const Singleton&);//声明即可
15         //2)使用静态成员变量维护单例对象
16         static Singleton s_instance;  <--------------+
17 public:                                               |
18         //3)提供单例对象的访问方法                    |
19         static Singleton& getInstance(void){         |
20                 return s_instance;                   |
21         }                                            |
22 private:                                            |
23         int m_data;                                  |
24 };                                                   |
25 Singleton Singleton::s_instance(123);---------------+
26 int main(void){
27         cout << "main 开始运行" << endl;
28         Singleton& s1 = Singleton::getInstance();
29         Singleton& s2 = Singleton::getInstance();
30         Singleton& s3 = Singleton::getInstance();
31         s1.print();
32         s2.print();
33         s3.print();
34         cout << "&s1:" << &s1 << endl;
35         cout << "&s2:" << &s2 << endl;
36         cout << "&s3:" << &s3 << endl;
37
38         //Singleton s4(321);//error
39         //Singleton* s5 = new Singleton(321);//error
40         //Singleton s6 = s1;//应该 error
41         //cout << "&s6:" << &s6 << endl;
42
43         return 0;
44 }
```

优点：代码实现简单、访问效率高、多线程安全
缺点：浪费内存

--》懒汉式：单例对象在用时再创建，不用即销毁

06lazy.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 //单例模式:懒汉式
 4 class Singleton{
 5 public:
 6     void print(void)const{
 7         cout << m_data << endl;
 8     }
 9 private:
10     //1)私有化构造函数(包括拷贝构造函数)
11     Singleton(int data=0):m_data(data){
12         cout << "单例对象被创建了" << endl;
13     }
14     Singleton(const Singleton&);
15     ~Singleton(void){
16         cout << "单例对象被销毁了" << endl;
17     }
18     //2)使用静态成员变量维护单例对象
19     static Singleton* s_instance;
20 public:
21     //3)提供单例对象的方法
22     static Singleton& getInstance(void){
23         if(s_instance == NULL){
24             s_instance = new Singleton(123);
25         }
26         ++s_count;
27         return *s_instance;
28     }
29     //单例对象不用即销毁,销毁时机?所有的使用者都不再使用才能销毁.
30     void release(void){
31         if(--s_count == 0){
32             delete s_instance;
33             s_instance = NULL;
34         }
35     }
36 private:
37     int m_data;
38     //计数:记录单例对象使用者的个数
39     static int s_count;
40 };
41 Singleton* Singleton::s_instance=NULL;
42 int Singleton::s_count = 0;
43
44 int main(void){
45     cout << "main 开始运行" << endl;
```

```
46      Singleton& s1 = Singleton::getInstance();//++s_count:1,new
47      Singleton& s2 = Singleton::getInstance();//++s_count:2
48      Singleton& s3 = Singleton::getInstance();//++s_count:3
49      s1.print();
50      s1.release();//--s_count:2
51
52      s2.print();
53      s3.print();
54      cout << "&s1:" << &s1 << endl;
55      cout << "&s2:" << &s2 << endl;
56      cout << "&s3:" << &s3 << endl;
57
58      s2.release();//--s_count:1
59      s3.release();//--s_count:0,delete
60
61      return 0;
62  }
```

07lazy_thread.cpp(线程安全懒汉式)

```
 1  #include <iostream>
 2  #include <pthread.h>
 3  using namespace std;
 4  //单例模式:懒汉式(线程安全)
 5  class Singleton{
 6  public:
 7      void print(void)const{
 8          cout << m_data << endl;
 9      }
10  private:
11      //1)私有化构造函数(包括拷贝构造函数)
12      Singleton(int data=0):m_data(data){
13          cout << "单例对象被创建了" << endl;
14      }
15      Singleton(const Singleton&);
16      ~Singleton(void){
17          cout << "单例对象被销毁了" << endl;
18      }
19      //2)使用静态成员变量维护单例对象
20      static Singleton* s_instance;
21  public:
22      //3)提供单例对象的方法
23      static Singleton& getInstance(void){
24          pthread_mutex_lock(&mutex);//加锁
25          if(s_instance == NULL){
26              s_instance = new Singleton(123);
27          }
28          ++s_count;
```

```
29              pthread_mutex_unlock(&mutex);//解锁
30              return *s_instance;
31      }
32      //单例对象不用即销毁,销毁时机?所有的使用者都不再使用才能销毁.
33      void release(void){
34              pthread_mutex_lock(&mutex);
35              if(--s_count == 0){
36                      delete s_instance;
37                      s_instance = NULL;
38              }
39              pthread_mutex_unlock(&mutex);
40      }
41 private:
42      int m_data;
43      //计数:记录单例对象使用者的个数
44      static int s_count;
45      //互斥锁
46      static pthread_mutex_t mutex;
47 };
48 Singleton* Singleton::s_instance = NULL;
49 int Singleton::s_count = 0;
50 pthread_mutex_t Singleton::mutex = PTHREAD_MUTEX_INITIALIZER;
51
52 int main(void){
53      cout << "main 开始运行" << endl;
54      Singleton& s1 = Singleton::getInstance();//++s_count:1,new
55      Singleton& s2 = Singleton::getInstance();//++s_count:2
56      Singleton& s3 = Singleton::getInstance();//++s_count:3
57      s1.print();
58      s1.release();//--s_count:2
59
60      s2.print();
61      s3.print();
62      cout << "&s1:" << &s1 << endl;
63      cout << "&s2:" << &s2 << endl;
64      cout << "&s3:" << &s3 << endl;
65
66      s2.release();//--s_count:1
67      s3.release();//--s_count:0,delete
68
69      return 0;
70 }
```

优点：节省内存
缺点：代码实现复杂、访问效率低、多线程需要加锁保护

十九 成员指针 //了解

1 成员变量指针(在类中的相对地址)

1）语法

类型 类名::* 成员指针变量名 = &类名::成员变量;

2）使用

对象.*成员指针变量名;//".*":被称为直接成员指针解引用操作符

对象指针->*成员指针变量名;//"->*":被称为间接成员指针解引用操作符

08memptr.cpp

```
#include <iostream>
#include <cstdio>
using namespace std;
class A{
public:
        A(int i=0):m_i(i){}
        int m_j;
        int m_i;
};
int main(void){
        //成员变量指针
        int A::*m_pi = &A::m_i;
        A a(100);
        A* pa = new A(200);
        cout << a.*m_pi << endl;
        cout << pa->*m_pi << endl;

        printf("&a=%p\n",&a);
        printf("m_pi=%p\n",m_pi);
        printf("&a.m_i=%p\n",&a.m_i);

        return 0;
}
```

2 成员函数指针

1）语法

返回类型 (类名::*成员函数指针)(参数表) = &类名::成员函数名;

2）使用

(对象.*成员函数指针)(实参表);

(对象指针->*成员函数指针)(实参表);

09memptr.cpp

```
#include <iostream>
using namespace std;
class A{
public:
        A(int i=0):m_i(i){}
        void print(void){
                cout << m_i << endl;
        }
```

```
 9 private:
10         int m_i;
11 };
12 int main(void){
13         //成员函数指针
14         void (A::*pfunc)(void) = &A::print;
15         A a(123);
16         A* pa = new A(321);
17         (a.*pfunc)();
18         (pa->*pfunc)();
19         return 0;
20 }
```

二十 操作符重载(operator)

1 双目操作符重载(L#R)

1.1 计算类双目操作符：+ - ...

--》表达式结果为右值，不能对表达式结果再赋值

--》左右操作数既可以为左值也可以为右值

--》两种实现方式：

1）成员函数形式(左调右参数)

L#R 的表达式将会被编译器处理为 L.operator#(R)成员函数调用形式，该函数的返回结果就是表达式结果。

2）全局函数形式(左右都为参)

L#R 的表达式也可以被编译器处理为 operator#(L,R)全局函数调用形式，该函数的返回结果就是表达式结果。

注：两种形式可以任选其一，但是不能都实现.

注：全局函数形式通常需要访问类中的私有成员，这时可以使用 friend 关键字将全局函数声明为相应类的友元，友元函数可以访问类中的任何成员。

01Complex.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 //复数
 4 class Complex{
 5 public:
 6         Complex(int r,int i):m_r(r),m_i(i){}
 7         void print(void)const{
 8                 cout << m_r << "+" << m_i << "i" << endl;
 9         }
10         //重载+:实现自定义的复数对象相加
11         //注:匹配的左右操作数既可以是左值也可以是右值,不能对表达式结果再赋值
12         //c1+c2 ==> c1.operator+(c2)
13         //三个 const 作用:
14         //1)修饰返回值,禁止对表达式结果再赋值
15         //2)常引用参数,支持常量型的右操作数
16         //3)常成员函数,支持常量型的左操作数
17         const Complex operator+(const Complex& c) const {
```

```
18              Complex res(m_r+c.m_r , m_i+c.m_i);
19              return res;
20      }
21      const Complex operator-(const Complex& c) const{
22              Complex res(m_r-c.m_r , m_i-c.m_i);
23              return res;
24      }
25 private:
26      int m_r;//实部
27      int m_i;//虚部
28 };
29 int main(void){
30      Complex c1(1,2);
31      Complex c2(3,4);
32      c1.print();//1+2i
33      c2.print();//3+4i
34      //Complex c3 = c1.operator+(c2);
35      Complex c3 = c1 + c2;
36      c3.print();//4+6i
37      //Complex c4 = c2.operator-(c1)
38      Complex c4 = c2 - c1;
39      c4.print();//2+2i
40
41      return 0;
42 }
```

02Complex.cpp

```
1 #include <iostream>
2 using namespace std;
3 //复数
4 class Complex{
5 public:
6      Complex(int r,int i):m_r(r),m_i(i){}
7      void print(void)const{
8              cout << m_r << "+" << m_i << "i" << endl;
9      }
10 private:
11      int m_r;//实部
12      int m_i;//虚部
13      //友元
14      friend const Complex operator+(
15              const Complex& left,const Complex& right);
16      friend const Complex operator-(
17              const Complex& left,const Complex& right);
18 };
19 //重载+/-:全局函数形式
20 const Complex operator+(const Complex& left,const Complex& right){
```

```
21      Complex res(left.m_r+right.m_r , left.m_i+right.m_i);
22      return res;
23  }
24  const Complex operator-(const Complex& left,const Complex& right){
25      Complex res(left.m_r-right.m_r , left.m_i-right.m_i);
26      return res;
27  }
28  int main(void){
29      Complex c1(1,2);
30      Complex c2(3,4);
31      c1.print();//1+2i
32      c2.print();//3+4i
33      //Complex c3 = operator+(c1,c2);
34      Complex c3 = c1 + c2;
35      c3.print();//4+6i
36      //Complex c4 = operator-(c2,c1)
37      Complex c4 = c2 - c1;
38      c4.print();//2+2i
39
40      return 0;
41  }
```

1.2 赋值类双目操作符: += -= ...
--》表达式结果是左值，就是左操作数的自身
--》左操作数一定是左值，右操作数既可以是左值也可以是右值
--》两种实现方式：
1）成员函数形式：L#R ==> L.operator#(R)
2）全局函数形式：L#R ==> operator#(L,R)

03Complex.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Complex{
4  public:
5      Complex(int r,int i):m_r(r),m_i(i){}
6      void print(void)const{
7          cout << m_r << "+" << m_i << "i" << endl;
8      }
9      //+=:成员函数形式
10     //c1+=c2 ==> c1.operator+=(c2);
11     Complex& operator+=(const Complex& c){
12         m_r += c.m_r;
13         m_i += c.m_i;
14         return *this;
15     }
16     //-=:全局函数形式(可以使用 friend 将其定义在类的内部,但本质还是全局函数)
17     //c1-=c2 ==> operator-=(c1,c2)
```

```
18      friend Complex& operator-=(Complex& left,const Complex& right){
19          left.m_r -= right.m_r;
20          left.m_i -= right.m_i;
21          return left;
22      }
23  private:
24      int m_r;//实部
25      int m_i;//虚部
26  };
27  int main(void){
28      Complex c1(1,2);
29      Complex c2(3,4);
30      c1 += c2;//c1.operator+=(c2);
31      c1.print();//4+6i
32      Complex c3(5,6);
33      (c1 += c2) = c3;//ok
34      c1.print();//5+6i
35
36      c1 -= c2;//operator-=(c1,c2)
37      c1.print();//2+2i
38      (c1 -= c2) = c3;
39      c1.print();//5+6i
40
41      return 0;
42  }
```

2 单目操作符重载 #O

2.1 计算类单目操作符：-(负) ~(位反)

--》表达式结果是右值，不能对表达式结果再赋值

--》操作数既可以是左值也可以是右值

--》两种实现方式：

1）成员函数形式：#O ==> O.operator#()

2）全局函数形式：#O ==> operator#(O)

```
04  1 #include <iostream>
    2 using namespace std;
    3 class Integer{
    4 public:
    5     Integer(int i=0):m_i(i){}
    6     void print(void) const {
    7         cout << m_i << endl;
    8     }
    9     //-(负):成员函数形式
   10     const Integer operator-(void) const {
   11         Integer res(-m_i);
   12         return res;
   13     }
```

```
14      //~:全局函数形式(自定义表示平方功能)
15      friend const Integer operator~(const Integer& i){
16          Integer res(i.m_i * i.m_i);
17          return res;
18      }
19  private:
20      int m_i;
21  };
22  int main(void){
23      Integer i(100);
24      Integer j = -i;//i.operator-()
25      j.print();//-100
26      j = ~i;//operator~(i);
27      j.print();//10000
28      return 0;
29  }
```

2.2 自增减单目操作符：前后++、--

1）前++、--

--》表达式结果是左值，就是操作数自身

--》操作数一定是左值

--》两种实现方式

成员函数形式：#O ==> O.operator#()

全局函数形式：#O ==> operator#(O)

2）后++、--

--》表达式结果是右值，是操作数自增减前的的数值

--》操作数一定是左值

--》两种实现方式

成员函数形式：O# ==> O.operator#(,int/*哑元*/)

全局函数形式：O# ==> operator#(O,int/*哑元*/)

注：后缀自增减为了和前缀区分，语法规定增加了哑元参数

05Integer.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Integer{
4  public:
5      Integer(int i=0):m_i(i){}
6      void print(void) const {
7          cout << m_i << endl;
8      }
9      //前++:成员函数形式,++i ==> i.operator++()
10     Integer& operator++(void){
11         ++m_i;
12         return *this;
13     }
```

```
14      //前--:全局函数形式,--i ==> operator--(i)
15      friend Integer& operator--(Integer& i){
16          --i.m_i;
17          return i;
18      }
19      //后++:成员函数形式,i++ ==> i.operator++(0/*哑元*/)
20      const Integer operator++(int/*哑元*/){
21          Integer old = *this;
22          ++*this;//++m_i;
23          return old;
24      }
25      //后--:全局函数形式,i-- ==> operator--(i,0/*哑元*/)
26      friend const Integer operator--(Integer& i,int/*哑元*/){
27          Integer old = i;
28          --i;//--i.m_i;
29          return old;
30      }
31  private:
32      int m_i;
33  };
34  int main(void){
35      Integer i(100);
36      Integer j = ++i;
37      i.print();//101
38      j.print();//101
39      j = ++++i;
40      i.print();//103
41      j.print();//103
42
43      j = --i;
44      i.print();//102
45      j.print();//102
46      j = ----i;
47      i.print();//100
48      j.print();//100
49
50      j = i++;
51      i.print();//101
52      j.print();//100
53
54      j = i--;
55      i.print();//100
56      j.print();//101
57
58      return 0;
59  }
```

3 输出和输入操作符重载:<< >>
功能：实现自定义类型对象的直接输出或输入
注：只能选择全局函数形式

```
#include <iostream>
ostream //标准输出流类，cout 就是该类的对象
istream //标准输入流类，cin 就是该类的对象

//cout<<a ==> operator<<(cout,a)
friend ostream& operator<<(ostream& os,const Right& right){
    ...
    return os;
}

//cin>>a ==> operator>>(cin,a)
friend istream& operator>>(istream& is,Right& right){
    ...
    return is;
}
```

06io_operator.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class Student{
 4 public:
 5     Student(const string& name,int age,int no)
 6         :m_name(name),m_age(age),m_no(no){}
 7     //重载<<,cout<<s ==> operator<<(cout,s)
 8     friend ostream& operator<<(ostream& os,const Student& s){
 9         os << "我叫" << s.m_name << ",今年" << s.m_age << "岁,学号"
10             << s.m_no;
11         return os;
12     }
13     //重载>>,cin>>s ==> operator>>(cin,s)
14     friend istream& operator>>(istream& is,Student& s){
15         cout << "请输入姓名:";
16         is >> s.m_name;
17         cout << "请输入年龄:";
18         is >> s.m_age;
19         cout << "请输入学号:";
20         is >> s.m_no;
21         return is;
22     }
23 private:
24     string m_name;
25     int m_age;
26     int m_no;
27 };
```

```
28 int main(void){
29     Student stu("张飞",26,10011);
30     cout << stu << endl;
31     Student stu2("赵云",24,10012);
32     cout << stu << ";" << stu2 << endl;
33
34     Student stu3("",0,0);
35     cin >> stu3;
36     cout << stu3 << endl;
37     return 0;
38 }
```

练习：实现 3*3 矩阵类，支持如下操作符重载

+ - += -= -(负) 前后++、-- <<

* *=

提示：

```
class M33{
public:
    ...
private:
    int m_a[3][3];
};
M33 m1(..);//1 2 3 4 5 6 7 8 9
M33 m2(..);//9 8 7 6 5 4 3 2 1
```

```
m1        +        m2          =   ?
1 2 3         9 8 7          10 10 10
4 5 6           6 5 4    =     10 10 10
7 8 9           3 2 1          10 10 10
m1        -        m2          =   ?
1 2 3           9 8 7          -8 -6 -4
4 5 6           6 5 4    =     -2  0  2
7 8 9           3 2 1           4  6  8
m1        *        m2          =   ?
1 2 3           9 8 7          30  24  18
4 5 6           6 5 4    =     84  69  54
7 8 9           3 2 1          138 114 90
```

07MM3.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class M33{
 4 public:
 5     M33(void){
 6         for(int i=0;i<3;i++)
 7             for(int j=0;j<3;j++)
 8                 m_a[i][j] = 0;
 9     }
10     explicit M33(int a[][3]){
```

```
11          for(int i=0;i<3;i++)
12              for(int j=0;j<3;j++)
13                  m_a[i][j] = a[i][j];
14      }
15      //重载<<
16      friend ostream& operator<<(ostream& os,const M33& m){
17          for(int i=0;i<3;i++){
18              for(int j=0;j<3;j++){
19                  os << m.m_a[i][j] << ' ';
20              }
21              cout << endl;
22          }
23          return os;
24      }
25      //重载+
26      const M33 operator+(const M33& m) const {
27          int a[3][3] = {0};
28          for(int i=0;i<3;i++)
29              for(int j=0;j<3;j++)
30                  a[i][j] = m_a[i][j] + m.m_a[i][j];
31          //M33 res(a);
32          //return res;
33          return M33(a);
34      }
35      //重载-
36      const M33 operator-(const M33& m) const {
37          int a[3][3] = {0};
38          for(int i=0;i<3;i++)
39              for(int j=0;j<3;j++)
40                  a[i][j] = m_a[i][j] - m.m_a[i][j];
41          M33 res(a);
42          return res;
43      }
44      //重载*
45      const M33 operator*(const M33& m) const {
46          int a[3][3] = {0};
47          for(int i=0;i<3;i++)
48              for(int j=0;j<3;j++)
49                  for(int k=0;k<3;k++)
50                  a[i][j] += m_a[i][k] * m.m_a[k][j];
51          M33 res(a);
52          return res;
53      }
54      //重载+=
55      //"m1+=m2" <=等价=> "m1=m1+m2"
56      M33& operator+=(const M33& m){
57          *this = *this + m;
```

```
58              return *this;
59          }
60          //-=
61          M33& operator-=(const M33& m){
62              *this = *this - m;
63              return *this;
64          }
65          //*=
66          M33& operator*=(const M33& m){
67              *this = *this * m;
68              return *this;
69          }
70          //-(负)
71          const M33 operator-(void) const {
72              //M33 m;
73              //return m - *this;
74              return M33() - *this;
75          }
76          //前++
77          M33& operator++(void){
78              for(int i=0;i<3;i++)
79                  for(int j=0;j<3;j++)
80                      ++m_a[i][j];
81              return *this;
82          }
83          //前--
84          M33& operator--(void){
85              for(int i=0;i<3;i++)
86                  for(int j=0;j<3;j++)
87                      --m_a[i][j];
88              return *this;
89          }
90          //后++
91          const M33 operator++(int/*哑元*/){
92              M33 old = *this;
93              ++*this;
94              return old;
95          }
96          //后--
97          const M33 operator--(int/*哑元*/){
98              M33 old = *this;
99              --*this;
100             return old;
101         }
102
103 private:
104         int m_a[3][3];
```

```
105 };
106 int main(void){
107     int a1[3][3] = {1,2,3,4,5,6,7,8,9};
108     M33 m1(a1);
109     int a2[3][3] = {9,8,7,6,5,4,3,2,1};
110     M33 m2(a2);
111     cout << m1 << endl;
112     cout << m2 << endl;
113     cout << "m1+m2:" << endl;
114     cout << m1+m2 << endl;
115     cout << "m1-m2:" << endl;
116     cout << m1-m2 << endl;
117     cout << "m1*m2:" << endl;
118     cout << m1*m2 << endl;
119
120     cout << "m1+=m2:" << endl;
121     cout << (m1+=m2) << endl;
122     cout << m1 << endl;
123     cout << "m1-=m2:" << endl;
124     cout << (m1-=m2) << endl;
125     cout << m1 << endl;
126     cout << "m1*=m2:" << endl;
127     cout << (m1*=m2) << endl;
128     cout << m1 << endl;
129
130     cout << "-m2:" << endl;
131     cout << -m2 << endl;
132
133     cout << "++m2:" << endl;
134     cout << ++m2 << endl;
135     cout << m2 << endl;
136
137     cout << "--m2:" << endl;
138     cout << --m2 << endl;
139     cout << m2 << endl;
140
141     cout << "m2++" << endl;
142     cout << m2++ << endl;
143     cout << m2 << endl;
144
145     cout << "m2--" << endl;
146     cout << m2-- << endl;
147     cout << m2 << endl;
148
149     return 0;
150 }
```

4 下标操作符重载 []
功能：实现让自定义类型的对象能够像数组一样去使用
注：非常对象返回左值，常对象返回右值

```
string s = "hello";
s[0] = 'H'; //s.operator[](0)
------------------------------
const string& rs = s;
cout << rs[0] << endl;//'H'
```

08Array.cpp

```
#include <iostream>
using namespace std;
//容器类:里面可以存放若干个 int 数据
class Array{
public:
	Array(size_t size){
		m_arr = new int[size];
	}
	~Array(void){
		delete[] m_arr;
	}
	//重载[]
	int& operator[](size_t i){//返回左值
		return m_arr[i];
	}
	const int& operator[](size_t i)const{//返回右值
		return m_arr[i];
	}
private:
	int* m_arr;
};
int main(void){
	Array arr(5);
	arr[0] = 123;//arr.operator[](0) = 123
	const Array& carr = arr;
	//carr[0] = 321;//error
	cout << carr[0] << endl;//123,ok
	return 0;
}
```

5 函数调用操作符重载 ()
功能：实现让自定义类型对象能够像函数一样去使用（仿函数）
注：对参数个数、类型和返回类型没有任何限制

```
A a;
a(123,4.56);//a.operator()(123,4.56);
```

09Func.cpp

```
#include <iostream>
using namespace std;
class Func{
public:
	int operator()(int a,int b){
		return a * b;
	}
	int operator()(int a){
		return a * a;
	}
};
int main(void){
	Func func;
	//func.operator()(100,200)
	cout << func(100,200) << endl;//20000
	//func.operator()(200)
	cout << func(200) << endl;//40000
	return 0;
}
```

6 动态内存管理操作符重载 new/delete //了解

```
class 类名{
	static void* operator new(size_t size){...}
	static void operator delete(void* pv){...}
};
```

10new.cpp

```
#include <iostream>
#include <cstdlib>
using namespace std;
class A{
public:
	A(void){ cout << "A 的构造函数" << endl; }
	~A(void){ cout << "A 的析构函数" << endl; }
	static void* operator new(size_t size){
		cout << "分配内存" << endl;
		void* pv = malloc(size);
		return pv;
	}
	static void operator delete(void* pv){
		cout << "释放内存" << endl;
		free(pv);
	}
};
int main(void){
	//1)A* pa = (A*)A::operator new(sizeof(A))
	//2)pa->构造函数
```

```
21      A* pa = new A;
22
23      //1)pa->析构函数
24      //2)A::operator delete(pa)
25      delete pa;
26
27      return 0;
28  }
```

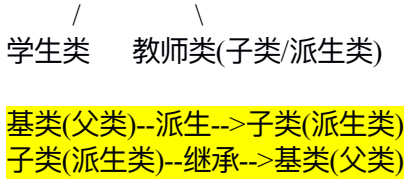
7 操作符重载的限制
1）不是所有的操作符都可以重载，下列几个不能重载
--》作用域限定操作符 "::"
--》直接成员访问操作符 "."
--》直接成员指针解引用操作符 ".*"
--》条件操作符 "?:"
--》字节长度操作符 "sizeof"
--》类型信息操作符 "typeid" //后面讲
2）如果一个操作符所有的操作数都是基本类型，则该操作符无法被重载
3）操作符重载不会改变编译器预定义的优先级
4）操作符重载不能改变操作数的个数
5）无法通过操作符重载机制发明新的操作符
6）只能使用成员函数形式实现不能使用全局函数(友元)形式实现的操作符
= [] () ->

二十一 继承(Inheritance)

1 继承的概念//了解
通过一种机制表达类型之间共性和特性的方式，利用已有的数据类型定义新的数据类型，这种机制就是继承.

eg：
人类：姓名、年龄、吃饭、睡觉
学生类：姓名、年龄、吃饭、睡觉、学号、学习
教师类：姓名、年龄、吃饭、睡觉、工资、讲课
...

人类：姓名、年龄、吃饭、睡觉
学生类继承人类：学号、学习
教师类继承人类：工资、讲课
...



基类(父类)--派生-->子类(派生类)
子类(派生类)--继承-->基类(父类)

2 继承语法
class 子类:继承方式 基类 1,继承方式 基类 2,...{

```
        ...
};
```

继承方式:

-- 公有继承(public)(struct 默认)

-- 保护继承(protected)

-- 私有继承(private)(class 默认)

01inherit.cpp

```
#include <iostream>
using namespace std;
class Human{//人类(基类)
public:
    Human(const string& name,int age):m_name(name),m_age(age){}
    void eat(const string& food){
        cout << "我在吃" << food << endl;
    }
    void sleep(int time){
        cout << "我睡了" << time << "小时" << endl;
    }
protected:
    //保护成员:在当前类的内部和子类中可以使用
    string m_name;
    int m_age;
};
class Student:public Human{//学生类(人类派生的子类)
public:
    //Human(name,age):指明基类部分的初始化方式
    Student(const string& name,int age,int no):Human(name,age),m_no(no){}
    void learn(const string& course){
        cout << "我在学" << course << endl;
    }
    void who(void){
        cout << "我叫" << m_name << ",今年" << m_age << "岁,学号是" <<
            m_no << endl;
    }
private:
    int m_no;
};
class Teacher:public Human{//教师类(人类派生的子类)
public:
    Teacher(const string& name,int age,int salary)
        :Human(name,age),m_salary(salary){}
    void teach(const string& course){
        cout << "我在讲" << course << endl;
    }
    void who(void){
        cout << "我叫" << m_name << ",今年" << m_age << "岁,工资为" <<
```

```
40                      m_salary << endl;
41          }
42  private:
43          int m_salary;
44  };
45  int main(void){
46          Student s("关羽",30,10086);
47          s.who();
48          s.eat("面包");
49          s.sleep(8);
50          s.learn("孙武兵法");
51          Teacher t("孙悟空",35,50000);
52          t.who();
53          t.eat("桃子");
54          t.sleep(6);
55          t.teach("unix 系统编程");
56
57          //Student*-->Human*:向上造型
58          Human* ph = &s;
59          ph->eat("面包");
60          ph->sleep(8);
61          //ph->who();
62
63          //Human*-->Student*:向上造型(合理)//安全
64          Student* ps = static_cast<Student*>(ph);
65          ps->who();
66
67          Human h("张飞",25);
68          //Human*-->Student*:向下造型(不合理)//危险
69          Student* ps2 = static_cast<Student*>(&h);
70          ps2->who();
71
72          return 0;
73  }
```

3 公有继承的特性

1）子类对象会继承基类的属性和行为，通过子类对象可以访问基类中的成员，如同是基类对象在访问它们一样。

注：子类对象中包含的基类部分被称为"基类子对象".

2）向上造型(upcast)//重点

将子类类型的指针或引用转换为基类类型的指针或引用；这种操作性缩小的类型转换，在编译器看来是安全的,可以直接隐式完成。

```
基 类
  ↑
子 类
-------------------------------------------------
class A{};
```

```
class B:public A{};
class C:public A{};
...
void func1(A* pa){}
void func2(A& ra){}
int main(void){
        B b;
        func1(&b);//向上造型
        func2(b);//向上造型
        C c;
        func1(&c);//向上造型
        func2(c);//向上造型
        return 0;
}
```

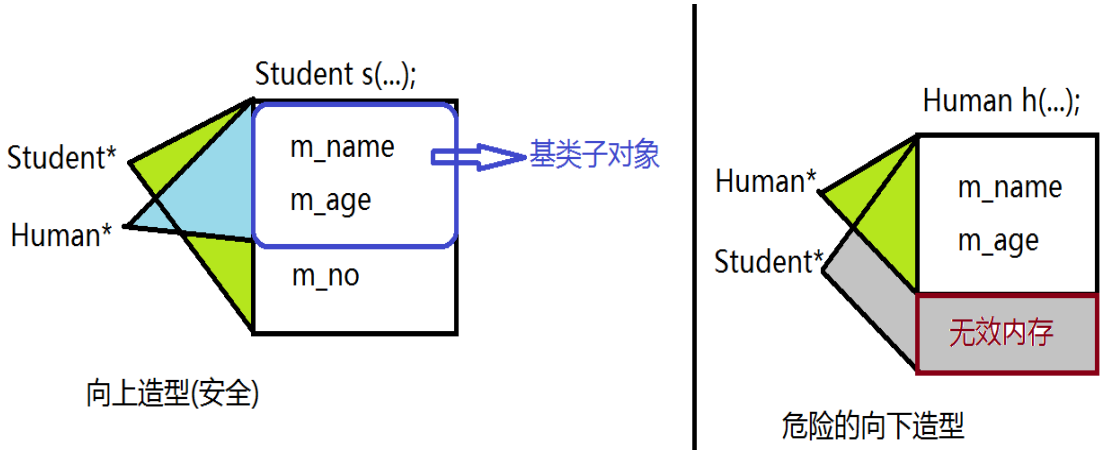
3）向下造型转换(downcast)

将基类类型的指针或引用转换为子类类型的指针或引用；这种操作性放大的类型转换，在编译器看来是危险的，不能隐式转换，但是可以显式转换(推荐使用 static_cast).

基 类

↓

子 类



4）子类继承基类的成员

--》在子类中，可以直接访问从基类中继承的公有成员和保护成员，就如同访问子类自己的成员一样.

--》基类的私有成员子类也可以继承过来，但是会受到访问控制属性的限制，无法直接访问；如果希望让子类访问到基类的私有成员，可以让基类提供公有或保护的成员函数，来间接访问。

02inherit.cpp

```
1 #include <iostream>
2 using namespace std;
3 class Base{//基类
4 public:
5       int m_public;
6 protected:
```

```
    int m_protected;
private:
    int m_private;
protected:
    //基类私有在子类中无法直接访问,但是可以提供类似如下的成员函数来间接访问
    void setPrivate(int data){
        m_private = data;
    }
    const int& getPrivate(void){
        return m_private;
    }
};
class Derived:public Base{//子类
public:
    void func(void){
        m_public = 10;
        m_protected = 20;
        cout << m_public << endl;
        cout << m_protected << endl;
        setPrivate(30);
        cout << getPrivate() << endl;
    }
};
int main(void){
    Derived d;
    d.func();
    cout << "size=" << sizeof(d) << endl;//12
    return 0;
}
```

5）子类隐藏基类的成员

--》如果子类和基类定义了同名的成员函数，因为作用域不同，不会构成重载关系，而是一种隐藏关系，这时通过子类对象将会优先访问子类自己的成员；这时如果还希望访问基类中所隐藏的成员，可以通过"类名::"来显式指明 //推荐

--》如果形成隐藏关系的成员函数参数不同，也可以通过 using 声明，将基类中的成员函数引入到子类的作用域，让它们形成重载，通过函数的重写匹配来解决

03inherit.cpp

```
#include <iostream>
using namespace std;
class Base{
public:
    void func(void){
        cout << "Base::func" << endl;
    }
};
class Derived:public Base{
```

```
public:
    void func(int i){
        cout << "Derived::func" << endl;
    }
    //通过 using 声明,将基类的 func 引入到子类作用域,让它们形成重载
    //using Base::func;
};
int main(void){
    Derived d;
    d.Base::func();
    d.func(123);
    return 0;
}
```

4 访问控制属性和继承方式

1）访问控制属性：影响类中成员的访问位置

访问控制限定符	访问控制属性	内部访问	子类访问	外部访问	友元访问
public	公有成员	ok	ok	ok	ok
protected	保护成员	ok	ok	no	ok
private	私有成员	ok	no	no	ok

2）继承方式：影响通过子类访问基类中成员的可访问性

基类中的	在公有继承的子类变成	在保护继承的子类变成	在私有继承的子类变成
公有成员	公有成员	保护成员	私有成员
保护成员	保护成员	保护成员	私有成员
私有成员	私有成员	私有成员	私有成员

04inherit.cpp

```
#include <iostream>
using namespace std;
class A{
public:
    int m_public;//公有成员
protected:
    int m_protected;//保护成员
private:
    int m_private;//私有成员
};
class B:public A{//公有继承的子类
    void func(void){
        m_public = 123;//ok
        m_protected = 123;//ok
        //m_private = 123;//no
    }
};
```

```
18 class C:protected A{//保护继承的子类
19     void func(void){
20         m_public = 123;//ok
21         m_protected = 123;//ok
22         //m_private = 123;//no
23     }
24 };
25 class D:private A{//私有继承的子类
26     void func(void){
27         m_public = 123;//ok
28         m_protected = 123;//ok
29         //m_private = 123;//no
30     }
31 };
32
33 //通过子类的子类访问基类中的成员
34 class X:public B{
35     void foo(void){
36         m_public = 123;//ok
37         m_protected = 123;//ok
38         //m_private = 123;//no
39     }
40 };
41 class Y:public C{
42     void foo(void){
43         m_public = 123;//ok
44         m_protected = 123;//ok
45         //m_private = 123;//no
46     }
47 };
48 class Z:public D{
49     void foo(void){
50         //m_public = 123;//no
51         //m_protected = 123;//no
52         //m_private = 123;//no
53     }
54 };
55
56 int main(void){
57     //通过子类对象访问基类中的成员
58     B b;
59     b.m_public = 123;//ok
60     //b.m_protected = 123;//no
61     //b.m_private = 123;//no
62
63     C c;
64     //c.m_public = 123;//no
```

```
65      //c.m_protected = 123;//no
66      //c.m_private = 123;//no
67
68      D d;
69      //d.m_public = 123;//no
70      //d.m_protected = 123;//no
71      //d.m_private = 123;//no
72
73      return 0;
74  }
```

注：向上造型的语法在私有继承和保护继承中不再适用。

05upcast.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Base{
4  public:
5      int m_public;
6  };
7  //class Derived:public Base{};
8  //class Derived:protected Base{};
9  class Derived:private Base{};
10 int main(void){
11     Derived d;
12     //Base* pb = static_cast<Base*>(&d);//向上造型
13     //Base& rb = static_cast<Base&>(d);//向上造型
14     return 0;
15 }
```

5 子类的构造函数

1）如果子类的构造函数没有显式指明基类子对象的初始化方式，那么编译器将会自动调用基类的无参构造函数来初始化基类子对象.

2）如果希望基类子对象以有参的方式被初始化，则需要使用初始化列表显式指明需要的构造实参.

06constructor.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Member{
4  public:
5      Member(void):m_j(0){
6          cout << "Member(void)" << endl;
7      }
8      Member(int j):m_j(j){
9          cout << "Member(int)" << endl;
10     }
11     int m_j;
```

```
12 };
13 class Base{
14 public:
15     Base(void):m_i(0){
16         cout << "Base(void)" << endl;
17     }
18     Base(int i):m_i(i){
19         cout << "Base(int)" << endl;
20     }
21     int m_i;
22 };
23 class Derived:public Base{
24 public:
25     Derived(void){
26         cout << "Derived(void)" << endl;
27     }
28     //Base(i):指明基类子对象的初始化方式
29     //m_member(j):指明成员子对象的初始化方式
30     Derived(int i,int j):Base(i),m_member(j){
31         cout << "Derived(int)" << endl;
32     }
33     Member m_member;//成员子对象
34 };
35 int main(void){
36     Derived d1;
37     cout << d1.m_i << "," << d1.m_member.m_j << endl;//0,0
38     Derived d2(100,200);
39     cout << d2.m_i << "," << d2.m_member.m_j << endl;//100,200
40     return 0;
41 }
```

3）子类对象的创建过程(笔试题)
--》分配内存
--》构造基类子对象(按继承表顺序)
--》构造成员子对象(按声明的顺序)
--》执行子类构造函数代码

6 子类的析构函数
1）子类的析构函数，无论是自己定义的，还是编译器缺省提供的，都会自动调用基类的析构函数，完成基类子对象的销毁操作。
2）子类对象的销毁过程(笔试题)
--》执行子类的析构函数代码
--》析构成员子对象(按声明的逆序)
--》析构基类子对象(按继承表逆序)
--》释放内存
3）基类的析构函数不能调用子类的析构函数，所以如果 delete 一个指向子类对象的基类指针，实际被调用将只是基类的析构函数，子类的析构函数执行不到，有内存泄漏的风险！

```
class A{...};
class B:public A{...};
A* pa = new B;//pa 指向子类对象的基类指针
delete pa;//只会调用基类的析构函数，有内存泄漏的风险
```
解决方法:虚析构函数//后面讲

07destructor.cpp

```
#include <iostream>
using namespace std;
class Member{
public:
    Member(void):m_j(0){
        cout << "Member(void)" << endl;
    }
    Member(int j):m_j(j){
        cout << "Member(int)" << endl;
    }
    ~Member(void){
        cout << "~Member(void)" << endl;
    }
    int m_j;
};
class Base{
public:
    Base(void):m_i(0){
        cout << "Base(void)" << endl;
    }
    Base(int i):m_i(i){
        cout << "Base(int)" << endl;
    }
    ~Base(void){
        cout << "~Base(void)" << endl;
    }
    int m_i;
};
class Derived:public Base{
public:
    Derived(void){
        cout << "Derived(void)" << endl;
    }
    //Base(i):指明基类子对象的初始化方式
    //m_member(j):指明成员子对象的初始化方式
    Derived(int i,int j):Base(i),m_member(j){
        cout << "Derived(int)" << endl;
    }
    ~Derived(void){
        cout << "~Derived(void)" << endl;
```

```
41      }
42      Member m_member;//成员子对象
43  };
44  int main(void){
45      /*Derived d1;
46      cout << d1.m_i << "," << d1.m_member.m_j << endl;//0,0
47      Derived d2(100,200);
48      cout << d2.m_i << "," << d2.m_member.m_j << endl;//100,200
49      */
50
51      Base* pb = new Derived;//向上造型
52      //...
53      delete pb;//内存泄露(笔试题)解决方法后面讲
54
55      return 0;
56  }
```

7 子类拷贝构造和拷贝赋值

7.1 子类的拷贝构造

1）如果子类没有定义拷贝构造函数，编译器会为子类提供缺省的拷贝构造函数，该函数会自动调用基类的拷贝构造函数，完成基类子对象对象的拷贝初始化。

2）如果子类自己定义拷贝构造函数，需要使用初始化列表，显式指明基类子对象也要以拷贝的方式进行初始化。

```
class Base{};
class Derived:public Base{
    //Base(that):显式指明基类子对象以拷贝的方式进行初始化
    Derived(const Derived& that):Base(that),...{}
};
```

7.2 子类的拷贝赋值(操作符赋值)

1）如果子类没有定义拷贝赋值函数，那么编译器会为子类提供一个缺省的拷贝赋值函数，该函数会自动调用基类拷贝赋值函数，完成基类子对象的赋值。

2）如果子类自己定义了拷贝赋值函数，那么需要显式调用基类的拷贝赋值函数，完成基类子对象的赋值。

```
class Base{};
class Derived:public Base{
    Derived& operator=(const Derived& that){
        if(&that != this){
            //显式调用基类的拷贝赋值函,完成基类子对象的复制
            Base::operator=(that);
        }
        return *this;
    }
};
```

08copy.cpp

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

```
 3 class Base{
 4 public:
 5       Base(void):m_i(0){}
 6       Base(int i):m_i(i){}
 7       Base(const Base& that):m_i(that.m_i){
 8            cout << "基类的拷贝构造函数" << endl;
 9       }
10       Base& operator=(const Base& that){
11            cout << "基类的拷贝赋值函数" << endl;
12            if(&that != this){
13                 m_i = that.m_i;
14            }
15            return *this;
16       }
17       int m_i;
18 };
19 class Derived:public Base{
20 public:
21       Derived(void):m_j(0){}
22       Derived(int i,int j):Base(i),m_j(j){}
23       //Base(that):指明基类子对象以拷贝的方式进行初始化
24       Derived(const Derived& that):Base(that),m_j(that.m_j){}
25       Derived& operator=(const Derived& that){
26            if(&that != this){
27                 //显式调用基类的拷贝赋值函数,完成基类子对象的复制
28                 Base::operator=(that);
29                 m_j = that.m_j;
30            }
31            return *this;
32       }
33       int m_j;
34 };
35 int main(void){
36       Derived d1(100,200);
37       cout << d1.m_i << "," << d1.m_j << endl;//100,200
38       Derived d2(d1);//拷贝构造
39       cout << d2.m_i << "," << d2.m_j << endl;//100,200
40       Derived d3;
41       d3 = d1;//拷贝赋值,d3.operator=(d1)
42       cout << d3.m_i << "," << d3.m_j << endl;//100,200
43       return 0;
44 }
```

8 子类的操作符重载//了解

09operator.cpp

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

```
 2 using namespace std;
 3 class Human{
 4 public:
 5       Human(const string& name,int age):m_name(name),m_age(age){}
 6       friend ostream& operator<<(ostream& os,const Human& h){
 7            os << h.m_name << "," << h.m_age;
 8            return os;
 9       }
10 private:
11       string m_name;
12       int m_age;
13 };
14 class Student:public Human{
15 public:
16       Student(const string& name,int age,int no):Human(name,age),m_no(no){}
17       friend ostream& operator<<(ostream& os,const Student& s){
18            os << (Human&)s << "," << s.m_no;
19       }                    |&防止放到临时变量里,这样可以自动匹配到基类中的输出操作符重载
20 private:
21       int m_no;
22 };
23 int main(void){
24       Student s("张三",20,10001);
25       cout << s << endl;
26       return 0;
27 }
```

9 多重继承

1）一个子类可以同时继承多个基类，这样继承方式称为多重继承

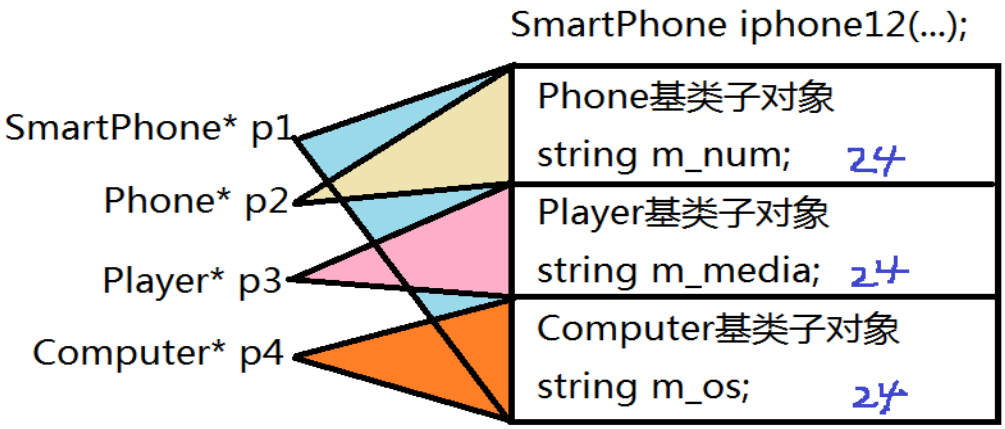
```
电话  播放器  计算机
   \      |     /
     智能手机
```

01mul_inherit.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class Phone{//电话基类
 4 public:
 5       Phone(const string& num):m_num(num){}
 6       void call(const string& num){
 7            cout << m_num << "打给" << num << endl;
 8       }
 9 private:
10       string m_num;
11 };
12 class Player{//播放器基类
13 public:
```

```
14      Player(const string& media):m_media(media){}
15      void play(const string& music){
16          cout << m_media << "播放器播放" << music << endl;
17      }
18  private:
19      string m_media;
20  };
21  class Computer{//计算机基类
22  public:
23      Computer(const string& os):m_os(os){}
24      void run(const string& app){
25          cout << "在" << m_os << "系统上运行" << app << endl;
26      }
27  private:
28      string m_os;
29  };
30  //智能手机子类
31  class SmartPhone:public Phone,public Player,public Computer{
32  public:
33      SmartPhone(const string& num,const string& media,const string& os)
34          :Phone(num),Player(media),Computer(os){}
35  };
36  int main(void){
37      SmartPhone iphone12("18866668888","MP3","Android");
38      iphone12.call("12315");
39      iphone12.play("海阔天空");
40      iphone12.run("王者荣耀");
41
42      SmartPhone* p1 = &iphone12;
43      Phone* p2 = p1;//向上造型
44      Player* p3 = p1;
45      Computer* p4 = p1;
46
47      cout << "p1=" << p1 << endl;
48      cout << "p2=" << p2 << endl;
49      cout << "p3=" << p3 << endl;
50      cout << "p4=" << p4 << endl;
51
52      SmartPhone* p5 = static_cast<SmartPhone*>(p4);//向下造型
53      cout << "p5=" << p5 << endl;
54
55      //cout << sizeof(string) << endl;//24
56      return 0;
57  }
```

2）向上造型时，编译器会根据各个基类子对象的内存布局，进行适当的偏移计算，以保证指针或引用的类型与其指向的基类子对象类型一致。



参考 mul_inherit.png(重点)

3）名字冲突问题
--》如果一个子类所继承的多个基类中存在相同名字，当通过子类访问这些名字时，编译器会报歧义错误--名字冲突.
--》解决名字冲突的通用做法就是显式的使用“类名::”,指明所访问的成员属于哪个基类//推荐
--》如果冲突的名字的是成员函数，并且参数不同，也可以通过 using 声明，让它们在子类中形成重载，通过函数重载的参数匹配来解决.

02mul_inherit.cpp

```
#include <iostream>
using namespace std;
class Base1{
public:
	void func(void){
		cout << "Base1 的 func" << endl;
	}
	int m_i;//成员变量
};
class Base2{
public:
	void func(int i){
		cout << "Base2 的 func" << endl;
	}
	typedef int m_i;//成员类型(int 类型别名)
};
class Derived:public Base1,public Base2{
public:
	//通过 using 声明,将两个基类的 func 都引入到子类作用域,让它们形成重载
	//using Base1::func;
	//using Base2::func;
};
int main(void){
```

```
24      Derived d;
25      d.Base1::func();
26      d.Base2::func(123);
27
28      d.Base1::m_i = 100;
29      Derived::Base2::m_i num = 200;
30      cout << d.Base1::m_i << "," << num << endl;//100,200
31      return 0;
32  }
```

10 钻石继承和虚继承

1）一个子类的多个基类源自公共的基类祖先，这样的继承结构称为钻石继承

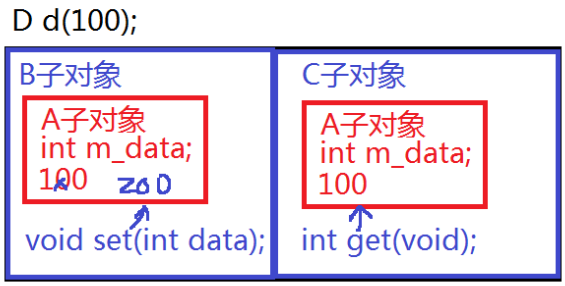
```
      A
     / \
    B   C
     \ /
      D
```

03diamond.cpp

```
1  #include <iostream>
2  using namespace std;
3  /* 钻石继承
4   *          A          //公共基类
5   *         / \
6   *        B   C        //中间类
7   *         \ /
8   *          D          //末端子类(关注)
9   * */
10 class A{
11 public:
12      A(int data):m_data(data){
13          cout << "A:" << this << ",size=" << sizeof(A) << endl;
14      }
15 protected:
16      int m_data;
17 };
18 class B:public A{
19 public:
20      B(int data):A(data){
21          cout << "B:" << this << ",size=" << sizeof(B) << endl;
22      }
23      void set(int data){
24          m_data = data;
25      }
26 };
27 class C:public A{
28 public:
```

```
29      C(int data):A(data){
30          cout << "C:" << this << ",size=" << sizeof(C) << endl;
31      }
32      int get(void){
33          return m_data;
34      }
35  };
36  class D:public B,public C{
37  public:
38      D(int data):B(data),C(data){
39          cout << "D:" << this << ",size=" << sizeof(D) << endl;
40      }
41  };
42  int main(void){
43      D d(100);
44      //cout << "sizeof(D):" << sizeof(d) << endl;//8
45      cout << d.get() << endl;//100
46      d.set(200);
47      cout << d.get() << endl;//200?100
48      return 0;
49  }
```

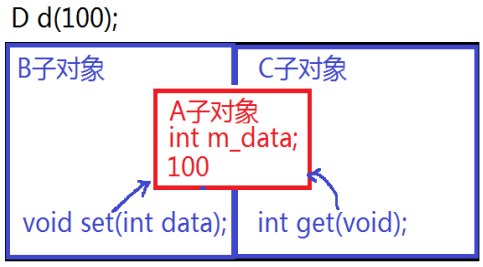
3）在创建末端子类(D)对象时，会包含多份公共基类(A)子对象；再通过末端子类访问公共基类中的成员，会因为继承路径不同，导致结果不一致。



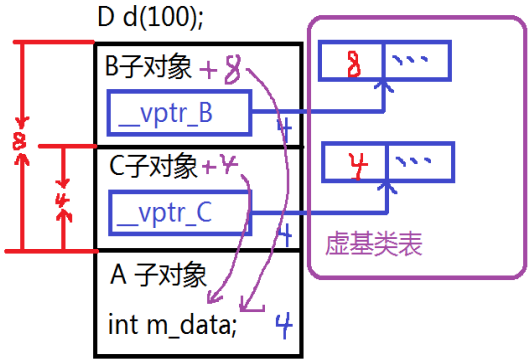
参考 diamond.png

3）通过虚继承可以让公共基类(A)子对象在末端子类对象中实例唯一，并为所有的中间类共享，这样即使沿着不同的继承路径，所访问到公共基类的成员也一定是一致的。

4）虚继承语法



钻石继承--虚继承



虚继承原理(了解)

参考 diamond.png

04diamond.cpp

```
#include <iostream>
using namespace std;
/* 钻石继承--虚继承
 *          A          //公共基类
 *        / \
 *       B   C         //中间类
 *        \ /
 *         D          //末端子类(关注)
 * */
class A{
public:
        A(int data):m_data(data){
                cout << "A:" << this << ",size=" << sizeof(A) << endl;
        }
protected:
        int m_data;
};
class B:virtual public A{//虚继承
public:
        B(int data):A(data){
                cout << "B:" << this << ",size=" << sizeof(B) << endl;
        }
        void set(int data){
                m_data = data;
        }
};
class C:virtual public A{//虚继承
public:
        C(int data):A(data){
                cout << "C:" << this << ",size=" << sizeof(C) << endl;
        }
        int get(void){
```

```
33              return m_data;
34          }
35  };
36  class D:public B,public C{
37  public:
38          //虚继承时,由当前末端子类负责构造公共基类子对象
39          D(int data):B(data),C(data),A(data){
40              cout << "D:" << this << ",size=" << sizeof(D) << endl;
41          }
42  };
43  int main(void){
44          D d(100);
45          //cout << "sizeo(D):" << sizeof(d) << endl;//8
46          cout << d.get() << endl;//100
47          d.set(200);
48          cout << d.get() << endl;//200
49          return 0;
50  }
```

--》在继承表中使用 virtual 关键字修饰
--》由继承链的末端子类负责构造公共基类子对象.

```
          A(int data)
         / \
        B   C  //class B/C:virtual public A
         \ /
          D    //负责构造公共基类子对象.
```

注：当使用虚继承语法时，在创建 D 对象，其继承结构类似如下:

```
          B   C   A
           \ | /
             D
```

二十二 多态(polymorphic)

案例：实现图形库，用于绘制各种图形

```
                    图形基类(位置/绘制)
                  /                    \
矩形(宽和高/绘制)                      圆形(半径/绘制)
```

05shape.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Shape{//图形基类
4  public:
5          Shape(int x=0,int y=0):m_x(x),m_y(y){}
6          virtual void draw(void){//虚函数
```

```
 7              cout << "绘制图形:" << m_x << "," << m_y << endl;
 8      }
 9  protected:
10      int m_x;//x 坐标
11      int m_y;//y 坐标
12  };
13  class Rect:public Shape{//矩形子类
14  public:
15      Rect(int x,int y,int w,int h):Shape(x,y),m_w(w),m_h(h){}
16      void draw(void){//自动变成虚函数
17              cout << "绘制矩形:" << m_x << "," << m_y << "," << m_w << ","
18                   << m_h << endl;
19      }
20  private:
21      int m_w;//宽
22      int m_h;//高
23  };
24  class Circle:public Shape{//圆形子类
25  public:
26      Circle(int x,int y,int r):Shape(x,y),m_r(r){}
27      void draw(void){//自动变成虚函数
28              cout << "绘制圆形:" << m_x << "," << m_y << "," << m_r << endl;
29      }
30  private:
31      int m_r;//半径
32  };
33  void render(Shape* buf[]){
34      for(int i=0;buf[i]!=NULL;i++){
35          /* 正常通过指针调用成员函数,根据指针的本身类型调用;但如果调用的是虚
36           * 函数,不再根据指针的本身,而是根据实际指向的目标对象类型调用.*/
37          buf[i]->draw();
38      }
39  }
40  int main(void){
41      Shape* buf[1024] = {NULL};
42      buf[0] = new Rect(10,20,5,6);
43      buf[1] = new Circle(15,12,11);
44      buf[2] = new Rect(11,22,15,16);
45      buf[3] = new Rect(20,30,51,61);
46      buf[4] = new Circle(19,26,15);
47      buf[5] = new Rect(10,20,5,6);
48      render(buf);
49
50      return 0;
51  }
```

1 虚函数覆盖(函数重写)、多态的概念

1）如果将基类中某个成员函数声明为虚函数，那么其子类中与该函数具有相同原型的成员就也是虚函数，并且可以对基类中的版本形成覆盖，也可以称为函数重写。
2）满足虚函数覆盖后，这时通过指向子类对象的基类指针或者通过引用子类对象的基类引用，调用虚函数，实际被执行的将是子类中的覆盖版本，而不是基类中的原始版本，这样语法现象被称为多态。

```
class Base{
public:
    virtual void func(void){}//虚函数
};
class Derived:public Base{
    void func(void){}//自动变成虚函数
}
int main(void){
    Derived d;
    Base* pb = &d;//pb：指向子类对象的基类指针
    Base& rb = d;//rb：引用子类对象的基类引用
    pb->func();//Derived::func
    rb.func();//Derived::func
}
```

2 虚函数覆盖的条件
1）只有类中的成员函数才能声明为虚函数，而全局函数、构造函数、静态成员函数都不能声明为虚函数。
注：析构函数可以为虚函数(特殊，后面讲)
2）只有在基类中以 virtual 关键字修饰的成员函数才能作为虚函数被子类覆盖，而与子类中的 virtual 关键字无关。
3）虚函数在子类中的覆盖版本和基类中原始版本要具有相同的函数签名，即函数名、参数表(类型和个数)、常属性要保持一致。

06override.cpp

```
#include <iostream>
using namespace std;
class A{};
class B:public A{};
class Base{
public:
    virtual void func(void){
        cout << "基类的 func" << endl;
    }
    virtual /*A**/A& foo(void){
        cout << "基类的 foo" << endl;
    }
};
class Derived:public Base{
public:
    void func(void){
        cout << "子类的 func" << endl;
```

```
18      }
19      /*B**/B&  foo(void){
20          cout << "子类的 foo" << endl;
21      }
22 };
23 int main(void){
24      Derived d;
25      Base* pb = &d;//pb:指向子类对象的基类指针
26      pb->func();
27      pb->foo();
28      return 0;
29 }
```

4）如果基类中虚函数返回基本类型的成员变量，那么子类的覆盖版本必须返回相同的类型；但如果基类中的虚函数返回类类型的指针(A*)或引用(A&)，那么允许子类的覆盖版本返回其子类类型的指针(B*)或引用(B&)。

```
class A{};
class B:public A{};
```

3 多态的条件

1）多态的语法特性除了要满足虚函数覆盖条件，还必须通过指针或引用调用虚函数才能表现出来

2）调用虚函数的指针也可以是 this 指针，当通过子类对象调用基类中的成员函数时，this 将是一个指向子类对象的基类指针，再通过 this 去调用虚函数，同样可以表现多态的语法.//重点掌握

07ploymorphic.cpp

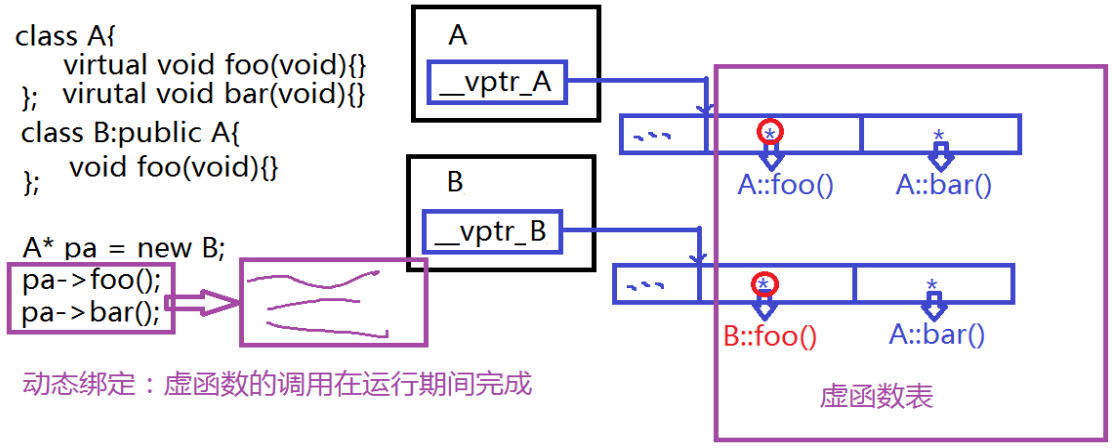
```
1 #include <iostream>
2 using namespace std;
3 class Base{
4 public:
5       virtual int cal(int x,int y){
6           return x + y;
7       }
8       //void func(Base* this)
9       void func(void){
10          //cout << this->cal(10,20) << endl;
11          cout << cal(10,20) << endl;//有多态现象
12      }
13 };
14 class Derived:public Base{
15 public:
16      int cal(int x,int y){
17          return x * y;
18      }
19 };
20 int main(void){
21      Derived d;
```

```
22      //Base b = d;
23      //cout << b.cal(10,20) << endl;//没有多态
24      d.func();//func(&d)
25      return 0;
26  }
```

eg：Qt 的多线程

```
class QThread{//官方写好的线程类
public:
    void start(void){
        //开启子线程，调用线程入口
        this->run();
    }
protected:
    virtual void run(void){
        //线程的入口函数
    }
};
class MyThread:public QThread{
protected:
    virtual void run(void){
        //重写线程的入口函数
    }
};
MyThread thread;
thread.start();//开启子线程，将会利用多态语法调用到子类重写的 run 函数
```

4 多态原理：多态语法通过"虚函数表"和"动态绑定"来实现//了解



参考 polymorphic.cpp

1）虚函数表会增加内存的开销
2）动态绑定会增加时间的开销
3）虚函数不能被内联优化
总结：实际开发中如果没有多态的要求，最好不要使用虚函数

01virtual.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class A{
 4 public:
 5       virtual void func(void){
 6             cout << "A::func" << endl;
 7       }
 8       virtual void foo(void){
 9             cout << "A::foo" << endl;
10       }
11 };
12 class B:public A{
13       void func(void){
14             cout << "B::func" << endl;
15       }
16       virtual void bar(void){
17             cout << "B::bar" << endl;
18       }
19 };//B 虚表:"B::func","A::foo","B::bar"???????????
20 int main(void){
21       B b;
22       cout << sizeof(b) << endl;
23       for(int i=0;i<3;i++){
24             (*(*(void(***)(void))&b+i))();//B::func,A::foo,B::bar
25       }
26 }
```

5 纯虚函数、抽象类和纯抽象类(笔试题)

1）纯虚函数

virtual 返回类型 函数名(参数表) = 0;//没有任何具体功能

2）抽象类

如果类中包含了纯虚函数，那么该类就是抽象类.

注：不能使用抽象类创建对象。

3）纯抽象类

如果类中所有的成员函数都是纯虚函数，那么该类就是纯抽象类，也可以称为接口类。

02shape.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class Shape{//图形基类,抽象类,纯抽象类
 4 public:
 5       Shape(int x=0,int y=0):m_x(x),m_y(y){}
 6       virtual void draw(void)=0;//纯虚函数
 7 protected:
 8       int m_x;//x 坐标
```

```
 9      int m_y;//y 坐标
10 };
11 class Rect:public Shape{//矩形子类
12 public:
13      Rect(int x,int y,int w,int h):Shape(x,y),m_w(w),m_h(h){}
14      void draw(void){//自动变成虚函数
15           cout << "绘制矩形:" << m_x << "," << m_y << "," << m_w << ","
16                << m_h << endl;
17      }
18 private:
19      int m_w;//宽
20      int m_h;//高
21 };
22 class Circle:public Shape{//圆形子类
23 public:
24      Circle(int x,int y,int r):Shape(x,y),m_r(r){}
25      void draw(void){//自动变成虚函数
26           cout << "绘制圆形:" << m_x << "," << m_y << "," << m_r << endl;
27      }
28 private:
29      int m_r;//半径
30 };
31 void render(Shape* buf[]){
32      for(int i=0;buf[i]!=NULL;i++){
33           /* 正常通过指针调用成员函数,根据指针的本身类型调用;但如果调用的是虚
34            * 函数,不再根据指针的本身,而是根据实际指向的目标对象类型调用.*/
35           buf[i]->draw();
36      }
37 }
38 int main(void){
39      Shape* buf[1024] = {NULL};
40      buf[0] = new Rect(10,20,5,6);
41      buf[1] = new Circle(15,12,11);
42      buf[2] = new Rect(11,22,15,16);
43      buf[3] = new Rect(20,30,51,61);
44      buf[4] = new Circle(19,26,15);
45      buf[5] = new Rect(10,20,5,6);
46      render(buf);
47
48      //Shape s;//error,抽象类不能创建对象
49
50      return 0;
51 }
```

03abstruct.cpp

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

```
 3 class PDFParser{
 4 public:
 5       void parse(const char* pdffile){
 6             cout << "解析出一行文本" << endl;
 7             onText();
 8             cout << "解析出一些图片" << endl;
 9             onImage();
10       }
11 private:
12       virtual void onText(void) = 0;
13       virtual void onImage(void) = 0;
14 };
15 class PDFRender:public PDFParser{
16 private:
17       void onText(void){
18             cout << "显示文本数据" << endl;
19       }
20       void onImage(void){
21             cout << "绘制图片数据" << endl;
22       }
23 };
24 int main(void){
25       PDFRender render;
26       render.parse("xx.pdf");
27       return 0;
28 }
```

6 虚析构函数(笔试题)

1）基类的析构函数不能调用子类的析构函数，所以如果 delete 一个指向子类对象的基类指针，实际被调用将只是基类的析构函数，子类的析构函数执行不到，有内存泄漏的风险！

//解决:虚析构函数

2）如果将基类的析构函数声明为虚析构函数，那么子类中的析构函数就也是虚析构函数，并且可以对基类中的虚析构函数形成有效的覆盖，可以表现多态的语法特性；这时再 delete 一个指向子类对象的基类指针，直接被执行的将是子类的析构函数，而子类析构函数又会自动调用基类的析构函数，从而避免了内存泄漏.

04vDestructor.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class Base{
 4 public:
 5       Base(void){
 6             cout << "基类动态资源分配" << endl;
 7       }
 8       virtual ~Base(void){//虚析构函数
 9             cout << "基类动态资源释放" << endl;
```

```
10      }
11  };
12  class Derived:public Base{
13  public:
14      Derived(void){
15          cout << "子类动态资源分配" << endl;
16      }
17      ~Derived(void){//自动变成虚函数函数
18          cout << "子类动态资源释放" << endl;
19      }
20  };
21  int main(void){
22      Base* pb = new Derived;
23      //pb->虚析构函数:实际执行的子类的析构函数
24      delete pb;
25
26      return 0;
27  }
```

二十三 运行时类型信息 //了解

1 typeid 操作符

#include <typeinfo>

typeid(类型/对象);

1）返回 typeinfo 对象，用于描述类型信息,可以使用 name()成员函数返回字符串形式的类型信息。

2）typeinfo 提供了比较操作符重载，可以通过它们直接进行类型之间的比较操作，如果类型之间具有多态的继承关系，还可以利用多态的有语法特性确定实际的目标对象类型.

01typeid.cpp

```
1  #include <iostream>
2  #include <typeinfo>
3  using namespace std;
4  class A{ virtual void func(void){} };
5  class B:public A{ void func(void){} };
6  class C:public A{ void func(void){} };
7
8  void foo(A& ra){
9      if(typeid(ra) == typeid(B)){
10         cout << "针对 B 子类对象的处理" << endl;
11     }
12     else{
13         cout << "其它对象的处理" << endl;
14     }
15 }
16
17 int main(void){
18     int i = 100;
```

```
19      cout << typeid(int).name() << endl;
20      cout << typeid(i).name() << endl;
21      int *a1[5];
22      int (*a2)[5];
23      cout << typeid(a1).name() << endl;
24      cout << typeid(a2).name() << endl;
25      cout << typeid( void (*[3])(int) ).name() << endl;
26
27      B b;
28      foo(b);
29      C c;
30      foo(c);
31
32      return 0;
33  }
```

2 dynamic_cast 操作符

1）语法：

目标类型变量 = dynamic_cast<目标类型>(源类型变量)；

2）适用场景：主要用于具有多态语法特性的父子类指针或引用之间显式类型转换

注：动态类型转换过程中，会检查目标对象类型和期望转换的类型是否一致，如果一致转换成功，否则失败，如果转换是指针，返回 NULL 表示失败，如果转换是引用，抛出"bad_cast"异常表示失败。

02dynamic_cast.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  class A{ virtual void func(void){} };
5  class B:public A{ void func(void){} };
6  class C:public A{ void func(void){} };
7
8  int main(void){
9       B b;
10      A* pa = &b;
11      //静态类型转换
12      //B* pb = static_cast<B*>(pa);//合理
13      //C* pc = static_cast<C*>(pa);//不合理
14      //动态类型转换
15      B* pb = dynamic_cast<B*>(pa);//合理,ok
16      C* pc = dynamic_cast<C*>(pa);//不合理,返回 NULL,表示失败
17      cout << "pa=" << pa << endl;
18      cout << "pb=" << pb << endl;
19      cout << "pc=" << pc << endl;
20
21      A& ra = b;
22      B& rb = dynamic_cast<B&>(ra);//合理
```

```
23      C& rc = dynamic_cast<C&>(ra);//不合理,抛出"bad_cast"异常表示失败
24
25      return 0;
26  }
```

二十四 异常机制(Exception)

1 软件开发中的常见错误

1）语法错误
2）逻辑错误
3）功能错误
4）设计缺陷
5）需求不符
6）环境异常
7）操作不当

2 传统 C 语言的错误处理机制

1）通过返回值表示错误

优点：函数调用路径中的栈对象可以得到正确的析构，内存安全

缺点：错误处理流程麻烦，需要逐层返回值判断，代码臃肿

03error.cpp

```
1  #include <iostream>
2  #include <cstdio>
3  using namespace std;
4  class A{
5  public:
6      A(void){ cout << "A 的构造函数" << endl; }
7      ~A(void){ cout << "A 的析构函数" << endl; }
8  };
9  int func3(){
10     A a;
11     FILE* fp = fopen("xx.txt","r");
12     if(fp == NULL){
13         cout << "文件打开失败" << endl;
14         return -1;
15     }
16     cout << "func3 正常执行" << endl;
17     fclose(fp);
18     return 0;
19 }
20 int func2(){
21     A a;
22     if(func3()==-1){
23         return -1;
24     }
25     cout << "func2 正常执行" << endl;
26     return 0;
```

```
27 }
28 int func1(){
29         A a;
30         if(func2()==-1){
31                 return -1;
32         }
33         cout << "func1 正常执行" << endl;
34         return 0;
35 }
36 int main(void){
37         if(func1()==-1){
38                 return -1;
39         }
40         return 0;
41 }
```

2）通过远程跳转处理错误

优点：函数调用不需要逐层返回值判断，一步到位错误处理，代码精炼

缺点：栈对象失去被析构的机会，有内存泄漏的风险

04error.cpp

```
 1 #include <iostream>
 2 #include <cstdio>
 3 #include <csetjmp>
 4 using namespace std;
 5 jmp_buf g_env;
 6 class A{
 7 public:
 8         A(void){ cout << "A 的构造函数" << endl; }
 9         ~A(void){ cout << "A 的析构函数" << endl; }
10 };
11 int func3(){
12         A a;
13         FILE* fp = fopen("xx.txt","r");
14         if(fp == NULL){
15                 longjmp(g_env,-1);
16         }
17         cout << "func3 正常执行" << endl;
18         fclose(fp);
19         return 0;
20 }
21 int func2(){
22         A a;
23         func3();
24         cout << "func2 正常执行" << endl;
25         return 0;
26 }
27 int func1(){
```

```
28      A a;
29      func2();
30      cout << "func1 正常执行" << endl;
31      return 0;
32  }
33  int main(void){
34      if(setjmp(g_env) == -1){
35          cout << "文件打开失败" << endl;
36          return -1;
37      }
38      func1();
39      return 0;
40  }
```

3 C++异常机制

1）异常抛出

throw 异常对象；

注：异常对象可以是基本类型的变量，也可以是类类型对象

2）异常检测和捕获

```
try{
    可能引发异常的语句;
}
catch(异常类型 1){
    针对异常类型 1 的处理
}
catch(异常类型 2){
    针对异常类型 2 的处理
}
...
```

注：catch 子类根据检测到的异常对象类型自上而下的顺序匹配，而不是最优匹配，因此对子类类型异常捕获语句要写在前面，否则将会被基类类型的异常捕获语句提前截获.

05exception.cpp

```
1  #include <iostream>
2  #include <cstdio>
3  using namespace std;
4  class FileError{
5  public:
6      FileError(const string& filename,int line)
7          :m_filename(filename),m_line(line){
8          cout << "出错位置:" << m_filename << "," << m_line << endl;
9      }
10 private:
11     string m_filename;
12     int m_line;
```

```
};
class A{
public:
    A(void){ cout << "A 的构造函数" << endl; }
    ~A(void){ cout << "A 的析构函数" << endl; }
};
int func3(){
    A a;
    FILE* fp = fopen("xx.txt","r");
    if(fp == NULL){
        throw FileError(__FILE__, __LINE__);//抛出异常(类类型对象)
        throw -1;//抛出异常(基本类型)
    }
    cout << "func3 正常执行" << endl;
    fclose(fp);
    return 0;
}
int func2(){
    A a;
    func3();
    cout << "func2 正常执行" << endl;
    return 0;
}
int func1(){
    A a;
    func2();
    cout << "func1 正常执行" << endl;
    return 0;
}
int main(void){
    try{//检测异常
        func1();
    }
    catch(int ex){//捕获异常
        if(ex == -1){
            cout << "文件打开失败" << endl;
            return -1;
        }
    }
    catch(FileError& ex){
        cout << "File Open Error" << endl;
        return -1;
    }

    return 0;
}
```

06exception.cpp(笔试题)

```
#include <iostream>
using namespace std;
class ErrorBase{};
class ErrorDerived:public ErrorBase{};
int main(void){
    try{
        throw ErrorBase();
        //throw ErrorDerived();
    }
    //异常对象有父子类继承关系时,需要把子类类型 catch 语句写在前面
    catch(ErrorDerived& ex){
        cout << "处理子类类型的异常" << endl;
    }
    catch(ErrorBase& ex){
        cout << "处理基类类型的异常" << endl;
    }
    return 0;
}
```

4 函数的异常说明

1）可以在函数原型中增加异常说明，用于指明该函数执行的期间可能抛出的异常类型

返回类型 函数名(形参表) throw(异常类型表) { ...}

eg：

```
class MemoryError{};
class FileError{};
//说明 func 函数在执行期间可能会抛出 MemoryError 或 FileError 的异常对象
void func(void) throw(MemoryError,FileError) {...}
```

07funcExcept.cpp

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

class FileError{};
class MemoryError{};

//函数声明
void func(void) throw(MemoryError,FileError);
//函数定义
void func(void) throw(FileError,MemoryError) {
    throw FileError();
    //throw MemoryError();
    //throw -1;
}
int main(void){
    try{
        func();
```

```
18        }
19        catch(FileError& ex){
20              cout << "文件相关错误处理" << endl;
21        }
22        catch(MemoryError& ex){
23              cout << "内存相关错误处理" << endl;
24        }
25        catch(int ex){
26              cout << "整型数相关错误处理" << endl;
27        }
28
29        return 0;
30  }
```

2）函数的异常说明是一种承诺，表示该函数所抛出的异常不会超出说明的范围，如果抛出了异常说明以外的其它类型，则无法被函数调用者正常捕获，而将会被系统捕获，导致进程终止。
3）两种极端形式
--》不写函数异常说明，表示可以抛出任何异常
--》空异常说明，throw()，表示不会抛出异常
4）如果函数的声明和定义分开写，要保证异常说明类型一致，但是顺序无所谓
5）虚函数覆盖条件补充
　　如果基类中的虚函数带有异常说明，那么该函数在子类中的覆盖版本不能比基类中原始版本抛出更多的异常，否则将会因为"放松 throw 限定"而导致编译失败.

08vfuncExcept.cpp

```
 1  #include <iostream>
 2  using namespace std;
 3  class FileError{};
 4  class MemoryError{};
 5  class Base{
 6  public:
 7        virtual void func(void) throw(FileError,MemoryError){
 8              cout << "基类的 func" << endl;
 9        }
10  };
11  class Derived:public Base{
12  public:
13        void func(void) throw(FileError,MemoryError) {
14              cout << "子类的 func" << endl;
15        }
16  };
17  int main(void){
18        Derived d;
19        Base* pb = &d;
20        pb->func();
21        return 0;
22  }
```

5 标准异常类:exception

```
class exception{
public:
    exception() throw() { }
    virtual ~exception() throw();

        /** Returns a C-style character string describing the general cause
        *   of the current error.   */
    virtual const char* what() const throw;
};
```

---

eg：

```
    try{
            ...
    }
    catch(exception& ex){//匹配 exception 任何子类类型异常对象
        ex.what();//利用多态语法，执行到子类中的 what 函数
    }
```

09stdExcept.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class FileError:public exception{
 4 public:
 5     virtual ~FileError() throw() {}
 6     virtual const char* what(void) const throw() {
 7         cout << "针对文件相关错误处理" << endl;
 8         return "FileError";
 9     }
10 };
11 class MemoryError:public exception{
12 public:
13     virtual ~MemoryError() throw() {}
14     virtual const char* what(void) const throw() {
15         cout << "针对内存相关错误处理" << endl;
16         return "MemoryError";
17     }
18 };
19 int main(void){
20     try{
21         throw MemoryError();
22         throw FileError();
23         //new 失败抛出异常"std::bad_alloc"
24         char* pc = new char[0xffffffff];
25     }
26     catch(exception& ex){
```

```
27              cout << ex.what() << endl;
28      }
29      return 0;
30 }
```

6 构造函数和析构函数中的异常//了解

1）构造函数可以抛出异常，但是这样的对象将会被不完整构造，这样对象其析构函数不会再自动被调用和执行，因此在构造函数抛出异常之前，需要手动销毁调用之前所分配动态资源，避免内存泄漏。

2）析构函数最好不要抛出异常

10consExcept.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 class A{
 4 public:
 5      A(void):m_pi(new int){
 6          if("error"){
 7              cout << "动态内存释放" << endl;
 8              delete m_pi;
 9              throw -1;
10          }
11          //...
12      }
13      ~A(void){
14          cout << "动态内存释放" << endl;
15          delete m_pi;
16      }
17 private:
18      int* m_pi;
19 };
20 int main(void){
21      try{
22          A a;
23      }
24      catch(int ex){
25          cout << "捕获到异常:" << ex << endl;
26      }
27      return 0;
28 }
```

11destructorexception.cpp

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

```
 5 public:
 6       void  func(void){
 7             throw  -1;
 8       }
 9       ~A(void){
10             throw  -2;
11       }
12 };
13 int  main(void){
14       try{
15             A  a;
16             a.func();//throw  -1
17       }//throw  -2
18       catch(int  ex){
19             cout  <<  "捕获到异常:"  <<  ex  <<  endl;
20       }
21       return  0;
22 }
```

二十五 I/O 流 //了解

1 主要的 I/O 流类

2 格式化 I/O

1）格式化函数(成员函数)

```
cout  <<  10/3.0  <<  endl;//3.33333
cout.precision(10);
cout  <<  10/3.0  <<  endl;//3.333333333
```

12format.cpp

```
 1 #include  <iostream>
 2 using  namespace  std;
 3
 4 int  main(void){
 5       cout  <<  10/3.0  <<  endl;
 6       //格式化函数
 7       cout.precision(10);//设置浮点数精度
 8       cout  <<  10/3.0  <<  endl;
 9       cout  <<  "[";
10       cout.width(10);//设置域宽
11       cout.fill('$');//设置空白位置用$字符填充
12       cout.setf(ios::showpos);//显示正号
13       cout.setf(ios::internal);//数据靠右,符号靠左
14       cout  <<  10000  <<  "]"  <<  endl;
15
```

```
16        return 0;
17 }
```

2）流控制符(全局函数)

```
cout << 10/3.0 << endl;//3.33333
cout << setprecision(10)<<10/3.0 << endl;//3.333333333
```

13format.cpp

```
 1 #include <iostream>
 2 #include <iomanip>
 3 using namespace std;
 4
 5 int main(void){
 6        cout << 10/3.0 << endl;
 7        //流控制符                                                    +--------正整数形式
 8        cout << setprecision(10) << 10/3.0 << endl;  |                  +-----设置对齐方式
 9        cout << "[" << setw(10) << setfill('$') << showpos << internal
10             << 10000 << "]" << endl;
11
12        //设置域宽 8 个字符,十六进制,左对齐,显示十六进制标识"0x"
13        cout << setw(8) << hex << left << showbase << 100 << endl;//0x64
14
15        return 0;
16 }
```

3）字符串流

#include<strstream>//过时，不推荐使用

istrstream,ostrstream

#include<sstream>//推荐

istringstream//类似 sscanf()

ostringstream//类似 sprintf()

14string.cpp

```
 1 #include <iostream>
 2 #include <sstream>
 3 using namespace std;
 4
 5 int main(void){
 6        int i = 123;
 7        double d = 4.56;
 8        /* char buf[1024] = {0};
 9          * sprintf(buf,"%d %lf",i,d);*/
10        ostringstream oss;
11        oss << i << ' ' << d;
12        cout << oss.str() << endl;
13
```

```
14      /* char buf[] = "321 6.54";
15       * sscanf(buf,"%d %lf",&i2,&d2);*/
16      istringstream iss("321 6.54");
17      int i2 = 0;
18      double d2 = 0.0;
19      iss >> i2 >> d2;
20      cout << i2 << "," << d2 << endl;
21
22      return 0;
23  }
```

4）文件流

```
#include<fstream>
ifstream//类似 fscanf()
ofstream//类似 fprintf()
```

15file.cpp

```
 1  #include <iostream>
 2  #include <fstream>
 3  using namespace std;
 4
 5  int main(void){
 6      //写文件
 7      ofstream ofs("file.txt");//FILE* fp = fopen("file.txt","w")
 8      //fprintf(fp,"%d %lf",123,4.56);
 9      ofs << 123 << ' ' << 4.56 << endl;
10      ofs.close();//fclose(fp)
11
12      //读文件
13      ifstream ifs("file.txt");//FILE* fp = fopen("file.txt","r")
14      int i=0;
15      double d=0.0;
16      ifs >> i >> d;//fscanf(fp,"%d%lf",&i,&d);
17      ifs.close();//fclose(fp)
18      cout << "i=" << i << ",d=" << d << endl;
19
20      return 0;
21  }
```

5）二进制 I/O

```
//类似 fread()
istream & istream::read(char * buffer,streamsize num);
//类似 fwrite
ostream & ostream::write(const char * buffer,size_t num);
```

16last.cpp

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

```
 2 #include <fstream>
 3 using namespace std;
 4
 5 int main(void){
 6         ofstream ofs("last.txt");
 7         char wbuf[] = "C++终于结束了!";
 8         ofs.write(wbuf,sizeof(wbuf));
 9         ofs.close();
10
11         ifstream ifs("last.txt");
12         char rbuf[100] = {0};
13         ifs.read(rbuf,sizeof(rbuf));
14         cout << "读到数据:" << rbuf << endl;
15         ifs.close();
16         return 0;
17 }
```