计算机程序设计基础（C++）知识与方法

T$^{\mathrm{T}}$T

2024 年 5 月 22 日

目录

1 类和对象

1.1 面向对象的程序设计

C 语言程序设计是**面向过程的程序设计**，其可以概括为：

- 范式：程序 = 算法 + 数据结构；
- 思想：（1）将程序看作一个“事件”；
  （2）按照事件发展的过程将“程序”分解为多个子过程（函数）；
  （3）将每个过程转写为算法（函数）；
- 特点：（1）以算法（函数）为中心；
  （2）**算法与数据分离**；
  （3）数据都是公用的，一个函数可以使用任何一组数据，一组数据又能被多个函数所使用；
- 核心问题：算法（函数）设计和数据定义。

而 C++ 语言程序设计是**面向对象的程序设计**，其可以概括为：

- 范式：程序 = 对象 + 消息；
- 思想：（1）任何系统都是有若干对象组成；
  （2）对象间通过消息作用构成一个有序系统（软件）；
  （3）程序以数据（对象）为中心；
- 对象两要素：属性（数据）和行为（方法/函数）；
- 特点：**数据和函数捆绑**，即属性和行为是对象不可分两特征；
- 核心问题：对象的设计与封装。

1.2 对象的定义与访问

定义 1.2.1. 对象、类

任何具备属性（attribute）和行为（behavior）两种要素的事物都可看成**对象（object）**，对象是属性（数据）和行为（函数）的**封装**。

一系列对象的共同特征的描述称为一个**类（class）**。

1.2.1 类与对象的定义

类是对象的**抽象（abstraction）**，对象是类的**实例（instance）**。类与对象的关系类似于结构体类型和结构体变量的关系。在程序设计中，往往先声明结构体类型，再用它去定义若干结构体变量；同样，往往先声明类，再用类定义对象。类是抽象的，不占用内存；对象是具体的，占用存储空间。

类声明的基本格式为（注意分号）：

```
class <-类名->
{
public:
    /* 公用数据与函数的定义或声明 */
private:
    /* 私有数据与函数的定义或声明 */
};
```

声明了类后，用已经声明的类来定义对象的基本格式为：

```
<-类名-> <-对象名->;
```

其中，private 和 public 称为**成员访问限定符**，缺省限定符为 private。private 后的数据与函数只能被类内成员函数调用，public 后的数据与函数可以在类外调用。通常在 public 后只留类内数据的访问接口。例如

```
#include<iostream>
using namespace std;
class Time              //类的声明
{
public:
    void set_time();
    void show_time();
private:
```

```
    int hour;
    int minute;
    int second;
};
int main()
{
    Time t1;            //对象定义
    t1.set_time();      //消息调用
    t1.show_time();
    Time t2;            //对象定义
    t2.set_time();      //消息调用
    t2.show_time();
}
```

类的成员函数可以在类之外定义，即

```
class Time
{
public:
    void show_time()
    {
        cout<<hour<<endl;
        cout<<minute<<endl;
        cout<<second<<endl;
    }
private:
    int hour;
    int minute;
    int second;
};
```

等价于

```
class Time
{
public:
    void show_time();
private:
    int hour;
    int minute;
    int second;
};
void Time::show_time()
{
    cout<<hour<<endl;
    cout<<minute<<endl;
    cout<<second<<endl;
}
```

同样要注意分号的使用。

注 1.1. inline 函数

函数调用时需要一定的时间和空间的开销。C++ 提供一种提高效率方法，即在编译时将被调函数的代码直接展开到主调函数中，而不是在函数执行时将流程转出去。这种能嵌到主调函数中函数称内置函数。

申明为内置函数只需在函数首行的左端加一个关键字 inline 即可。在类体中定义的成员函数，如果没有循环递归控制结构，就会被编译系统自动视为 inline 函数，因此类内无需用 inline 申明内置成员函数，而类外则不能省去 inlien。

1.2.2 对象成员的访问

访问对象中的 public 成员有三种方法，分别是：

- 通过对象名和成员运算符访问，如

```
Time today;
today.set_time();
```

- 通过指向对象的指针访问，即

```
Time today;
Time *ptime = &today;
ptime->set_time();
```

- 通过对象的引用变量访问，即

```
Time today;
Time &anyday = today;
anyday.set_time();
```

1.3 类的构造函数与析构函数

1.3.1 构造函数

定义 1.3.1. 构造函数

构造函数（constructor）是与其所在类同名的一个特殊 public 成员函数，在定义对象时自动执行，专门用来处理类下对象的初始化。

若没有定义，则编译器在类中添加一个空构造函数，仍调用但不执行任何操作。

自定义构造函数有以下类型：

- **无参数构造函数**，使得每个新对象都有相同的初始值，定义格式为：

```
<-类名（构造函数名）->()
{
    /* 对数据变量初始化等 */
}
```

其中函数体也可以加入其他操作，但这可能会降低代码的复用性。

- **含参数构造函数**，使得每个新对象都依据定义时的参数取不同的初始值，此时类的声明与对象的定义基本格式为：

```
class <-类名->
{
public:
    <-类名（构造函数名）->(<-类型1-> <-形参1->, <-类型2-> <-形参2->, …)
    {
        /* 用形参对数据变量初始化等 */
    }
private:
    /* 数据变量表 */
}
int main()
{
    <-类名-> <-对象名-> (<-实参1->, <-实参2->, …);    //用类定义对象同时初始化
}
```

即在定义对象时直接在后面加上参数。

含参数的构造函数可以**重载**，即给一个函数名以不同的函数体以便对类对象提供不同的初始化方法。但对象构建时只能执行其一，此时只能用参数表区分，因此每个函数体的参数个数或参数类型必须有所不同，即根据定义时给的参数表确定执行的函数体。例如：

```
class Time
{
public:
    Time(int h, int m, int s) { hour = h, minute = m, second = s; }
    Time() { hour = 24, minute = 60, second = 60; }
private:
    int hour, minute, second;
};
```

```
int main()
{
    Time nowtime1;                  // 初始化为 24, 60, 60
    Time nowtime2(20, 26, 3);       // 初始化为 20, 26, 3
}
```

C++ 支持将格式相近的构造函数体合并，即使得参数带有默认值。例如，上例中构造函数即可写成：

```
Time(int h = 24, int m = 60, int s = 60) { hour = h, minute = m, second = s; }
```

默认参数的构造函数允许形参没有实参输入，因此不能再在对应位置重载构造函数，如上面的构造函数与任意参数数量不多于三个的函数体都不兼容。

C++ 对数据变量初始化的函数体提供了简写形式，即用冒号引出**参数初始化表**的写法。其基本格式为：

```
<-类名（构造函数名）->(<-类型1-> <-形参1->, <-类型2-> <-形参2->, …)
    : <-成员变量1->(<-初始值1->), <-成员变量2->(<-初始值2->), …
    {}
```

其中成员变量的初始值是形参的表达式，通常就是形参本身；形参可以带有自己的默认值。例如，下面写法与上例等价：

```
Time(int h = 24, int m = 60, int s = 60): hour(h), minute(m), second(s) {}
```

注 1.2. 关于构造函数的几点说明

（1）构造函数必须声明为 public 成员；

（2）构造函数的调用时间为对象进入作用域、对象的生命周期开始时刻，只能由编译器根据对象产生以隐藏方式调用，不能由程序代码直接调用；

（3）构造函数没有返回值，在声明和定义时不需要注明 void 返回类型；其函数体可以包含包括 cin、cout 语句在内的任何内容，但不提倡在构造函数中加入与初始化无关的操作；

（4）当且仅当用户没有定义构造函数，编译器自动提供缺省的无参数构造函数，其函数体为空。

1.3.2 析构函数

定义 1.3.2. 析构函数

析构函数（destructor）是以其所在类名称加上波浪线 ~ 前缀为名的一个特殊 public 成员函数，在对象生命期结束时自动执行，专门用来执行类下对象的善后工作。

注 1.3. 关于析构函数的几点说明

（1）析构函数必须声明为 public 成员；

（2）析构函数的调用时间为对象离开作用域或被删除等对象的生命期结束时刻，只能由编译器根据对象回收以隐藏方式调用，不能由程序代码直接调用；

（3）析构函数没有返回值和参数，在声明和定义时不需要注明 void 返回类型，且一个类只有一个析构函数，不能重载；其函数体可以包含包括 cin、cout 语句在内的任何内容，建议析构函数进行对象回收

时的内存清理等结束工作；

（4）当且仅当用户没有定义析构函数，编译器自动提供缺省的析构函数，其函数体为空。

注 1.4. 运算符 new 与 delete

new 运算符用于为构建对象申请内存空间，new CLASS 申请一块大小为类 CLASS 的对象大小的内存空间，同时对其按类 CLASS 调用构造函数。

delete 运算符用于释放对象的内存空间，delete pCLASS 释放 pCLASS 指向的对象的内存空间，同时对其按用 new 构建时的类 CLASS 调用析构函数。

1.3.3 拷贝构造函数

对象的**拷贝**，是指用一个对象的数据快速地产生出多个相同对象，可看做是对象的定义与赋值过程的结合。对象拷贝的基本格式可写成 <-类名-> <-新对象->(<-原对象->);，也可写成赋值形式的 <-类名-> <-新对象-> = <-原对象->;。

发生拷贝时，新对象的定义不会调用构造函数，而会调用**拷贝构造函数**，其声明格式为：

```
<-类名（拷贝构造函数名）-> (const <-类名-> &object);
```

其中形参 *object* 是本类的一个对象。用同样的方式可以自定义拷贝构造函数，但由于参数表已经确定，拷贝构造函数不能重载。事实上，拷贝构造函数就是构造函数的以类的对象的引用为参数的一个重载。

注 1.5. 隐式拷贝

除了在显式地基于已有对象定义新对象时会调用拷贝构造函数外，在对象作为函数参数或返回值时，还会发生调用拷贝构造函数的**隐式拷贝**。下面是一个例子，其中 *today* 作为实参传递给形参 *A*，*B* 作为返回值传递出函数体（因为执行函数和执行赋值表达式是前后两步，此时只能传递给一个临时变量 *temp*）都是调用拷贝构造函数的隐式拷贝。

```
Date func(Date A)
{
    Date B(A);
    return B;                  // 此处有返回值传出
}
void main()
{
    Date today(1,1,1900);
    today = func(today);       // 此处有参数值传入
}
```

1.4 对象的调用形式

1.4.1 对象的浅拷贝问题

若类的数据域中有指针域，将一个对象的各成员值赋给另一个对象的各成员时，指针类成员拷贝的只是地址，使得不同对象中的指针成员指向同一块内存，造成了隐含的空间共享，破坏了对象之间的独立性，就称为**浅拷贝**。

解决浅拷贝问题，需要重新编写拷贝构造函数，以对指针类成员重新分配空间，称为进行**深拷贝**。

例如，对下面的类 Strings：

```
class Strings
{
public:
    Strings(char *s);
    ~Strings();
    void Print();
    void Set(char *s);
private:
    int length;
    char *str;
};
```

直接使用默认的缺省拷贝构造函数，指针 *str* 会出现浅拷贝问题。因此，需要重新编写拷贝构造函数：

```
class Strings
{
public:
    Strings(char *s);
    Strings(Strings &p);
    ~Strings();
    void Print();
    void Set(char *s);
private:
    int length;
    char *str;
};
Strings::Strings(Strings &p)     // 新的拷贝构造函数
{
    length = strlen(p.str);
    str = new char[length + 1];
    strcpy(str, p.str);
}
```

1.4.2 对象数组

静态构建对象数组的基本格式为：

```
<-类名-> <-数组名->[<-数组长度->] = { <-初始参数表-> };
```

其中初始参数表中，数组各项的初始参数表用逗号隔开，写成调用构造函数的形式，即写成

```
<-类名-> <-数组名->[<-数组长度->] = {
    <-类名（构造函数名）->(<-参数表1->),
    <-类名（构造函数名）->(<-参数表2->), …
};
```

动态构建对象数组使用 new 运算符：

```
<-类名-> * <-指针名-> = new <-类名->[<-数组长度->];
```

其指针指向数组的第 0 个对象的首地址。

要释放使用 new 运算符动态构建的对象数组，则需要对应地使用：

```
delete [] <-指针名->;
```

注 1.6. 对象数组的释放

直接使用 delete <-指针名->; 也可以释放对象。但是，在释放对象数组时，此语句只能释放对象数组的首对象，没有对后面的对象执行析构函数；而失去首对象后，后面的对象也无法再以数组形式访问。

1.4.3 对象指针

访问对象中成员变量的指针与一般指向结构体成员指针类似。成员访问限定符并不能限制通过指针访问类的 private 成员变量。

访问对象中成员函数的指针指向对象中某个成员函数的入口地址，其和普通函数的指针变量定义方法不同，定义形式为：

```
<- 返回值数据类型 -> (<-类名->::*<-指针变量名->)(<-参数表->);
```

赋值形式为：

```
<-指针变量名-> = &<-类名->::<-成员函数名->;
```

调用形式为：

```
……(<-对象名->.*<-指针变量名->)(<-参数表->)……
```

成员函数指针的使用要注意三个方面的匹配，即

- 定义函数指针使用的参数表（类型和数目）与成员函数参数表匹配；
- 函数指针的返回值类型与成员函数返回值匹配；
- 函数指针所在的类与成员函数所在的类匹配。

同样，成员访问限定符并不能限制通过指针访问类的 private 成员函数。

注 1.7. 成员函数指针的对象绑定

在面向对象的程序设计，也即 C++ 程序设计中，函数总是与类绑定的。类的成员函数通过 *this* 指针调用对象中的数据，而其本身并不与对象绑定。因此，成员函数指针在定义乃至赋值阶段都不与对象绑定，只有在调用时才通过 *this* 指针的值与对象绑定。在定义、赋值时，函数指针都由类限定，只有调用时才由对象限定。

注 1.8. *this* 指针

每个成员函数都包含一个特殊的指针，即 *this*。*this* 的值是当前被调用的成员函数所在对象的起始地址。执行 *object*.function(); 语句时，即将 function() 的 *this* 赋为 &*object*。*this* 指针也可以作为返回值的构成部分，传出一个对象或其地址，如下面的例子。

```
class Date
{
public:
    Date & add_day();
    void Print();
private:
    /* 日期数据的成员变量 */
};
Date & Date::add_day()
{
    /* 实现日期加一日功能的代码 */
    return *this;
}
void main()
{
    Date today(1,1,1900);
    today.add_day.Print();
            // 此处 today.add_day 传出一个对象，接着调用该对象的 Print() 函数
}
```

1.5 共用数据的保护

1.5.1 const 常类型

表 1: 用 const 进行变量或函数的定义或声明

类型	语句	说明
常变量	`const <-数据类型-> <-变量名-> = <-初值->;`	定义自身内容不可更改的普通变量
常指针	`const <-数据类型->* <-指针变量名->;` `<-数据类型-> * const <-指针变量名-> = <-初值->;`	定义**指向内存空间不可更改**、自身指向地址可以更改的指针变量 定义**指向地址不可更改**的指针变量
常对象	`<-类名-> const <-对象名->(<-参数表->);`	定义自身内容不可更改的对象，其必须通过构造函数进行初始化；除了隐式调用的构造和析构函数以外，**不能调用常对象的非 const 成员函数**
常对象指针	`const <-类名->* <-指针变量名->;` `<-类名-> * const <-指针变量名-> = <-初值->;`	定义**指向的对象内存空间不可更改**，自身指向地址可以更改的类指针变量 定义**指向的地址不可更改**的类指针；其指针值始终保持为其初值，不可更改，即只能指向一个对象
常引用	`const <-类名-> & <-指针变量名-> = <-初值->;`	定义一个对象的只读引用
常成员变量	类声明中：`const <-数据类型-> <-变量名->;`	定义自身内容不可更改的成员变量，其必须**通过参数初始化表赋初值**
常成员函数	类声明中： `<-返回值类型-> <-函数名->(<-参数表->) const;` 类声明外：`<-返回值类型-> <-类名->::<-函数名->(<-参数表->) const {<-函数体->};`	声明一个函数体不能更改类成员变量的成员函数 定义一个函数体不能更改类成员变量的成员函数

表 2: const 对成员函数与成员变量作用的影响

数据成员类型	操作	非 const 成员函数	const 成员函数
非 const 对象的非 const 成员	引用（读）	可以	可以
	更改（写）	可以	不可以
非 const 对象的 const 成员	引用（读）	可以	可以
	更改（写）	不可以	不可以
const 对象的成员	引用（读）	不可以	可以
	更改（写）	不可以	不可以

注 1.9. 如何利用 const 共用数据保护机制

C++ 类对象的赋值、拷贝往往被常引用替代：

（1）为提高效率可通过引用来传递类对象，但引用会共享空间，若要确保传入的对象内容不被改变，可以使用常引用；

（2）如果返回值是一个对象，也可以通过返回引用来提高效率，若要确保返回的对象内容不被改变，可以返回常引用；

（3）如果要确保对象内容不被改变，比常对象更好的方法是对象的常引用。

如果要确保对象的成员内容不被改变，比常对象和常成员变量更好的方法是定义常成员函数，用常成员函数作为对象属性的只读访问接口；但需注意，对于常对象，除了隐式调用的构造和析构函数以外，不能调用常对象的非 const 成员函数。

1.5.2 静态成员

为实现类的不同成员之间的通信，C++ 引入了类的**静态成员**。静态成员由类的所有对象共享，常用于描述类的属性。

定义类的静态成员的基本格式为：

```
class <-类名->
{
    ......
    static <-数据类型-> <-变量名->;
    ......
};
```

类的静态成员属于整个类，不属于任何一个具体对象，所以它们和全局变量一样，在所有本类对象产生之前就被分配。类的静态成员的初始化应当独立于类的对象的初始化进行。

例如，对包含年月日数据的 Date 类，要将默认日期定义为静态成员，有三种定义方式：

- 直接使用 public 静态成员变量，定义为

```
class Date
{
private:
    int y, m, d;
public:
    static int dy, dm, dd;
    Date(int yi, int mi, int di) : y(yi), m(mi), d(di) {}
};
```

这样就可以直接用

```
int Date::dy = 1900;
int Date::dm = 1;
int Date::dd = 1;
```

初始化三个静态变量；

- 使用 public 静态对象封装上面的三个变量，定义为

```
class Date
{
private:
    int y, m, d;
public:
    static Date defaultDate;
    Date(int yi, int mi, int di) : y(yi), m(mi), d(di) {}
};
```

这样则用

```
Date Date::defaultDate(1900, 1, 1);
```

初始化这个静态成员对象；

- 定义**静态成员函数**专门用来初始化静态对象，即

```
class Date
{
private:
    int y, m, d;
    static Date defaultDate;
public:
    Date(int yi, int mi, int di) : y(yi), m(mi), d(di) {}
    static void setdefault(int sy, int sm, int sd)
    {
        defaultDate.y = sy;
        defaultDate.m = sm;
        defaultDate.d = sd;
    }
};
```

这样调用函数即可完成初始化。

注 1.10. 静态成员的特性

（1）由于静态成员实质上是全局变量，所以不随对象产生而分配，也不随对象销毁而释放，在类声明时已经分配内存，在 main() 函数结束后才释放；

（2）静态成员为所有类对象所共有，每一个类对象的成员函数都可以访问它们，但在类外访问 public 静态成员变量时，一般使用「<-类名->::<-静态成员名->」方式，因为静态成员属于整个类，而不是某个具体对象；

（3）静态数据成员的初始化不能在构造函数、初始化参数表中进行，只能在类外进行；

（4）静态成员函数没有 this 指针，不能访问非静态数据成员。

1.6 类的继承与派生

1.6.1 派生类的声明

声明派生类的一般形式为：

```
class <-派生类名-> : <-继承方式-> <-基类名->
{
    // 新增加的成员
};
```

其中「继承方式」分为 public、protected、private，可以省略，缺省值为 private。

表 3: 基类成员在派生类中的访问权限

继承方式	基类中成员访问限定		
	public	protected	private
public	public	protected	inaccessible
protected	protected	protected	inaccessible
private	private	private	inaccessible

在派生类构造时，派生类会接受基类**除构造、析构函数外全部的成员**，改变其访问属性，同时增加新成员、进行同名屏蔽替换。例如，对下面的 CAnimal 类：

```
class CAnimal
{
public:
    void GetWeight() { cin >> m_fweight; }
private:
    float m_fweight;
};
```

其函数 GetWeight() 是 public 成员，可以在类外通过对象访问；但对其派生类 CPig：

```
class CPig : private CAnimal
{
public:
    void GetPorkWeight() { cin >> m_fPorkWeight; }
private:
    float m_fPorkWeight;
};
```

由于继承方式是 private，函数 GetWeight() 成为其 private 成员，在类外通过对象不能访问。

派生类的继承方式会限制基类成员在派生类中的访问级别，对基类中 public 与 protected 成员，可用 **using <-基类名->::<-成员名->** 改变基类访问级别：

- 在 public 下进行 using 声明的成员，可在派生类外通过对象访问，但不能用基类对象访问，如上面 CAnimal 类的派生类 CPig，若改为：

```
class CPig : private CAnimal
{
public:
    using CAnimal::GetWeight;    // 将函数 GetWeight() 用作 public 成员
    void GetPorkWeight() { cin >> m_fPorkWeight; }
private:
```

```
7       float m_fPorkWeight;
8   };
```

则其可以在类外通过对象访问，即下面第 8 行代码可以运行：

```
1   #include <iostream>
2   using namespace std;
3   int main(void)
4   {
5       CAnimal animal;
6       animal.GetWeight();
7       CPig apig;
8       apig.GetPorkWeight();
9       apig.GetWeight();
10  }
```

此时，同时实现了访问本不可访问的 *apig.m_fweight*；

- 在 protected 下进行 using 声明的成员，可以继续派生下去；
- 在 private 下进行 using 声明的成员，不能在派生类外通过对象访问。

多级继承时，仍有按表 3 确定的访问权限的演变。

1.6.2 派生类的构造函数和析构函数

派生类的构造函数不能继承，这是因为其构造函数不仅要考虑基类继承数据成员初始化，还应当考虑派生类所增加数据成员的初始化。编写派生类的构造函数时，**首先要通过参数初始化表调用基类的构造函数，对基类数据成员初始化**，然后再继续用参数初始化表初始化新增数据成员，或在函数体中初始化新增数据成员。

调用构造函数时，首先调用基类的构造函数，然后调用派生类的构造函数的剩余初始化部分。若有多级派生，在参数初始化表中只需调用直接基类的构造函数，构造对象时通过直接基类的构造函数先调用间接基类的构造函数，然后调用直接基类的构造函数，最后调用派生类的构造函数的剩余初始化部分。

注 1.11. 编写派生类构造函数的特殊情况

（1）当不需要对派生类新增的成员进行任何初始化操作时，派生类构造函数的函数体可以为空，即构造函数是空函数。

（2）当基类没有定义任何构造函数，或者只是定义了无参构造函数时，派生类构造函数可以不写基类构造函数调用，此时 C++ 编译器将自动调用基类的无参构造函数（若无，则调用默认构造函数），无需传递参数；

（3）C++11 中，引入了**继承构造函数**语法，即用语句 **using <-基类名->::<-基类名->** 直接将基类的构造函数作为派生类的构造函数，同时不能继承基类构造函数的默认值。

同样，派生类不能继承基类的析构函数，也需要通过派生类的析构函数去调用基类的析构函数；同时，可以在派生类中根据需要定义自己的析构函数，用来对派生类中所增加的成员进行清理工作，而基类成员的清理工作仍然由基类的析构函数负责。

在执行派生类的析构函数时，调用的顺序与构造函数正好相反：先执行派生类自己的析构函数，对派生类新增加的成员进行清理，然后调用基类的析构函数，对基类进行清理。

1.6.3 多重继承的同名覆盖与虚基类

一个派生类有两个或多个基类，派生类从两个或多个基类中继承所需的属性，这种写法称为**多重继承**，其一般格式为：

```
class <-派生类名-> : <-基类1继承方式-> <-基类1名->, ..., <-基类n继承方式-> <-基类n名->
{
    // 新增加的成员
};
```

多重继承的派生类的构造函数中，只需在参数初始化表中列出各直接基类构造函数即可。直接基类构造函数会按照参数初始化表中的顺序调用。

多重继承最常见的问题，是继承的成员同名产生的**二义性（ambiguity）**问题，即同一个成员名可能既可指向基类（可能还不止一个基类）中的成员，也可能指向派生类的新增成员。为此，成员的调用遵循**同名覆盖规则**，即如果在定义派生类对象的模块中通过对象名访问同名成员而不加修饰，则访问的是派生类的成员，即派生类优先原则。

注 1.12. 同名覆盖与函数重载

成员函数只有在函数名和参数都相同才发生同名覆盖。只是函数名相同而参数不同为函数重载。

多重继承的另一个问题，是同时有多级派生产生的**冗余性（redundancy）**问题，即如果一个派生类有多个直接基类，而这些直接基类又有一个共同的基类，则在最终的派生类中会保留该间接共同基类数据成员的多份同名成员。为此，可将基类作为**虚基类**进行派生，使在继承间接相同基类时只保留一份成员，即同名成员在内存中只有一份拷贝。虚基类声明的一般格式为：

```
class <-直接派生类名-> : virtual <-虚基类1继承方式-> <-虚基类1名->, ..., virtual <-虚基类
    n继承方式-> <-虚基类n名->, <-其他基类1继承方式-> <-其他基类1名->, ..., <-其他基类n继
    承方式-> <-其他基类n名->
{
    // 新增加的成员
};
```

虚基类的声明不是在最终的派生类中对直接基类做声明，而是**在各直接基类中对间接基类做声明**。为了保证虚基类在最终的派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类，否则仍然会出现对基类的多次继承。

注 1.13. 虚基类的初始化

（1）如果在虚基类中定义了带参数的构造函数，则在其所有派生类（包括直接派生或间接派生的派生类）中，如果需要使用这一重载，都要通过**构造函数的参数初始化表**对虚基类进行初始化，如：

```
class A {
    A(int i){}
    /* 其他成员 */ };
class B : virtual public A {
    B(int i) : A(i){}
    /* 其他成员 */ };
class C : virtual public A {
    C(int i) : A(i){}
    /* 其他成员 */ };
```

其中，对虚基类的间接派生类，参数初始化表中须**由其直接调用间接基类的有参构造函数**，因为各直接基类（上例 B、C）调用虚基类的构造函数时可能会出现参数矛盾；同时，按语法，各直接基类（上例 B、C）的构造函数调用也**不能省略**，但编译器会忽略直接基类中对虚基类的构造函数的调用，即写为

```
class D : public B, public C {
    D(int i, int j) : A(i), B(i), C(j){}
    /* 其他成员 */ };
```

（2）这些构造函数的调用遵循如下顺序：

- 虚基类的构造函数在非虚基类之前调用；
- 同一层包含多个虚基类，则按申明顺序调用；
- 若虚基类由非虚基类派生而来，则要先调用更高级别的基类构造函数，再遵循前两点的顺序。

注 1.14. 类的派生与类的组合

；类的**组合**，是指类申明时包含另一个类对象作为数据成员。
类的组合和继承一样，都是有效地利用已有类的资源。但二者有着本质区别：继承是纵向（同一系族）的，组合是横向（不同系族）的。

1.6.4 基类与派生类的兼容规则

基类与派生类的兼容规则是指，在需要基类对象的地方，都可使用 public 派生类的对象来替代。也即，派生类对象可作为基类对象使用，但只能使用派生类中的「基类成员」。通过兼容规则，可以做到：

- 派生类对象向基类对象赋值；
- 派生类对象向基类对象的引用赋值，此时该引用与派生类对象的基类部分共享同一段存储单元；
- 基类对象的指针指向派生类对象。

2 消息

2.1 运算符重载

C++ 中的运算符只能对基本类型的数据进行操作，作用比较有限。通过在运算符上重载新的操作意义，就可以用运算符操作自定义的对象。

2.1.1 成员运算符函数

在**类声明中**对运算符重载的一般格式为：

```
<-返回类型-> operator<-运算符-> (<-参数表->) { <-函数体-> }
```

也即，对运算符进行重载，可看做是将 `operator<-运算符->` 这个函数重载给运算符。

注 2.1. 重载

重载，是用户根据自己需要，对 C++ 已有运算符或函数的功能重新赋予新的含义，使之“一词多义”的过程。函数和运算符都可以重载，但需要让程序明确何时使用哪一语义。为此，需要让各个重载的参数表（数目和类型）不同。

例如，现有类 Complex 中的复数相加函数 Complex::add()：

```
class Complex
{
private:
    double real, imag;
public:
    Complex add(Complex& c)
    {
        Complex cresult;
        cresult.real = real + c.real;
        cresult.imag = imag + c.imag;
        return cresult;
    }
};
```

将它作为函数原型，可以在类 Complex 中重载运算符「+」：

```
class Complex
{
private:
    double real, imag;
public:
    Complex operator +(Complex& c)
    {
        Complex cresult;
        cresult.real = real + c.real;
        cresult.imag = imag + c.imag;
        return cresult;
    }
```

```
};
```

此时，原先写成 *c3 = c1.add(c2);* 的语句就可以写成 *c3 = c1 + c2;*。

一般地，运算符重载的参数表有如下规定：

- **双目运算符**的重载声明为一元的成员函数（如上例），运算符前面的对象（如上面的 *c1*）就是调用运算符函数的对象（即上面.add 前面的对象），而运算符后面的对象（或运算符函数形参类型的数据变量）作为函数的实参；
- **前置单目运算符**的重载声明为零元的成员函数；若函数是对对象的成员进行操作，其返回值常为 **this*，如：

```
class Complex
{
private:
    double real, imag;
public:
    Complex operator ++()
    {
        real += 1;
        return *this;
    }
};
```

- **后置单目运算符**的重载声明为带有一个无名 int 参数的成员函数，如：

```
class Complex
{
private:
    double real, imag;
public:
    Complex operator ++(int)
    {
        Complex temp(*this);
        real += 1;
        return temp;
    }
};
```

上面两个例子保持了前置、后置 ++ 的原有语义，即前置 ++ 返回值为自增后的值，后置 ++ 返回值为自增前的值。习惯上，运算符重载只是扩大原有运算符的适用范围，不改变运算符的原有语义。

注 2.2. 运算符重载的其他规则

（1）重载不能改变运算符操作数的个数、优先级别、结合性；

（2）重载运算符的函数不能有默认的参数；

（3）重载运算符必须和用户定义的自定义类型的对象一起使用，其参数至少应有一个是类对象（或类对象的引用）；

（4）用于类对象的运算符必须重载；但有两个例外，即运算符「=」和「&」不必用户重载。

2.1.2 友元运算符函数

外部函数可以声明为类的**友元**，从而可以访问类的私有成员。友元函数的声明形式为在类的 public 限定符下写：

```
friend <-返回类型-> <-函数名-> (<-参数表->);
```

也即在函数声明前加关键字 friend。

双目运算符要重载为第一操作数不为类对象的运算符函数，就必须重载为友元函数，如：

```
class Complex
{
private:
    double real, imag;
public:
    friend Complex operator +(double r, Complex& c);
};
Complex operator +(double r, Complex& c)
{
    Complex cresult;
    cresult.real = r + c.real;
    cresult.imag = c.imag;
    return cresult;
}
```

其他运算符重载的函数也可以放到类外，然后在类中声明为友元函数；但这会破坏类的封装性，一般只会在重载的运算符第一操作数不为本类对象，或有操作数为其他类对象等必要情况下才使用。

2.1.3 运算符重载的应用实例

A) 流的提取、插入运算符 头文件 iostream 中，定义了输入流类 istream 和输出流类 ostream；cin 和 cout 分别是 istream 类和 ostream 类的对象，且是全局对象；流插入运算符「<<」和流提取运算符「>>」是位运算符「<<」「>>」在 istream 类和 ostream 类中的重载，用来输出和输入标准类型数据。

对自定义类型数据，要使得「<<」「>>」也可以用于输入输出，则需要再次重载，形式为：

```
istream & operator >> (istream &, <-自定义类名-> &);
ostream & operator << (ostream &, <-自定义类名-> &);
```

下面是一个例子。

```
#include <iostream>
using namespace std;
class Complex
{
private:
    double real, imag;
public:
    Complex() { real = 0; imag = 0; }
    Complex(double r, double i) { real = r; imag = i; }
    friend ostream & operator << (ostream &, Complex &);  // 声明 << 重载为友元函数
    friend istream & operator >> (istream &, Complex &);  // 声明 >> 重载为友元函数
```

```
};

ostream & operator << (ostream & output, Complex & c)     // << 对 Complex 类的重载定义
{
    output << "(" << c.real;
    if (c.imag >= 0) output << "+";                       // 虚部非负时加上加号
    output << c.imag << "i)";
    return output;                // 保持 << 可以连续输出，即前面的「cout << c」返回 cout
}
istream & operator >> (istream & input, Complex & c)      // >> 对 Complex 类的重载定义
{
    cout << "Input the real part and the imaginary part of a complex number: " << endl;
    input >> c.real >> c.imag;
    return input;                 // 保持 >> 可以连续输入，即前面的「cin >> c」返回 cin
}

int main()
{
    Complex c1, c2;
    cin >> c1 >> c2;
    cout << "c_1 = " << c1 << endl << "c_2 = " << c2 << endl;
    return 0;
}
```

B) 带有越界检测的下标运算符 下标运算符「[]」是 C++ 常用运算符，但 [] 没有定义下标越界的检测功能，而下标越界的错误很容易产生。因此，考虑通过运算符重载，对下标运算符增加一个越界检测功能。

例如，定义一个替代 int 类型数组的类：

```
class intVector
{
    int* data;
    int size;
public:
    intVector() { data = NULL; size = 0; }
    ~intVector() { if (data != NULL) delete[] data; }
    int & operator [](int n);
};
```

其下标运算符 [] 重载为

```
int & intVector::operator [](int n)
{
    assert(n >= 0 && n < size);
    return data[n];
}
```

其中，operator [] 返回值为 int 的引用，以使得其结果可以为左值。**assert() 断言函数**是头文件 assert.h 中的一个调试用函数，当其实参表达式的结果为 0 时断言失败、程序出错，会向标准输出设备打印一条错误信息，并调用 abort() 终止程序。

注 2.3. assert() 的使用

assert() 断言失败抛出的 error 信息格式为：

```
Assertion failed: <– expression –>, file <–__FILE__–>, line <–__LINE__–>
```

通过抛出的 error 信息，可以知道出错的表达式及其位置，因而 assert() 常用于在函数开始处检验传入参数的合法性，即做合法性判断，防止后面表达式出错。因而，最好每个 assert() 只检验一个条件，因为同时检验多个条件时，如果断言失败，无法直观的判断是哪个条件失败。

代码编辑中解决方案配置有 Debug 和 Release 两种模式，Debug 是程序员在测试代码期间使用的编译模式，Release 是软件交付的编译模式。在 Release 模式下不应该调用 assert()，否则会抛出一段看不到的提示信息，并毫无预警地终止程序。assert.h 中对此有一定准备：

```
#ifdef NDEBUG
    #define assert(expression) ((void)0)
#else
    _ACRTIMP void __cdecl _wassert(
        _In_z_ wchar_t const* _Message,
        _In_z_ wchar_t const* _File,
        _In_  unsigned     _Line
        );
    #define assert(expression) (void)(                                                \
        (!!(expression)) ||                                                           \
        (_wassert(_CRT_WIDE(#expression), _CRT_WIDE(__FILE__), (unsigned)(__LINE__)), \
            0) \
    )
#endif
```

当没有宏定义 NDEBUG 时才会抛出错误信息，而有宏定义 NDEBUG 时，assert(expression) 即不会执行。因此，在 assert() 中不能使用改变环境的语句，否则一旦有宏定义 NDEBUG 就会使程序运行遇到问题。

类似地，也可以将二维数组下标（用「()」括起，用逗号分隔）运算符重载：

```
class intMatrix
{
    int* data;
    int row, col;
public:
    intMatrix() { data = NULL; row = 0; col = 0; }
    ~intMatrix() { if (data != NULL) delete[] data; }
    int & operator ()(int r, int c);
};
int & intMatrix::operator ()(int r, int c)
{
    assert(r >= 0 && r < row && c >= 0 && c < col);
    return data[col * r + c];
}
```

C) 内存管理运算符 new、delete 运算符也可进行重载，其重载形式既可是类成员函数，也可是友元函数，但第 1 个参数类型必须是 size_t。如果类中没有定义 new 和 delete 的重载函数，那么会自动调用缺省 new 和 delete。

new 先调用 operator new 申请内存，再调用构造函数初始化内存；delete 先调用析构函数，再调用 operatordelete 释放内存。new[] 与 delete[] 都只会调用一次 operator new[]、operator delete[]，且是从前往后构造、从后往前析构。

2.2 数据类型转换

2.2.1 转换构造函数

转换构造函数以其他类型数据为参数，构建自身类的对象。转换构造函数只有一个参数，其格式与有参构造函数一致；事实上，**只有一个参数的构造函数均可视为转换构造函数**。转换构造函数会在形参与实参类型不相同时自动尝试调用，如

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

class Complex
{
private:
    double real, imag;
public:
    Complex() { real = 0; imag = 0; }
        // 无参构造函数
    Complex(double r) { real = r; imag = 0; }
        // 构造函数的重载，同时是 double 到 Complex 的转换构造函数
    Complex(double r, double i) { real = r; imag = i; }
        // 构造函数的重载
    friend Complex operator +(Complex c1, Complex c2);
        // 以 Complex 类对象为参的运算符函数
    friend ostream& operator <<(ostream& output, Complex c);
        // Complex 类对象输出函数
};
Complex operator +(Complex c1, Complex c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
ostream& operator <<(ostream& output, Complex c)
{
    output << "(" << c.real << ", " << c.imag << ")";
    return output;
}

int main()
{
    Complex c1(3, 4), c2(5), c3, c4, c5; int i = 2;
                        // c2 的构造调用构造函数第 2 个重载
    c3 = c1 + 2.5;      // 运算符重载调用转换构造函数，相当于 c3 = c1 + Complex(2.5)
    c4 = 2.5 + c2;      // 运算符重载调用转换构造函数，相当于 c4 = Complex(2.5) + c2
    c5 = 2.5 + i;       // 赋值语句调用转换构造函数，相当于 c5 = Complex(2.5 + i)
    cout << c1 << " " << c2 << " " << c3 << " " << c4 << " " << c5 << " " << endl;
```

```
    return 0;
}
```

其输出为

```
(3, 4) (5, 0) (5.5, 4) (7.5, 0) (4.5, 0)
```

2.2.2 类型转换运算符

标准类型的类型名可以用于将其他标准类型数据强制转换为该类型数据，此为 C 中的强制转换运算符。沿用这一思路，类型名也可以用作将某类对象转换为该类型的运算符。类型转换运算符函数的重载形式为：

```
operator <-类型名-> () { /* 实现转换的语句 */ }
```

例如：

```
#include <iostream>
#include <cmath>                // hypot() 函数所需
using namespace std;

class Complex
{
private:
    double real, imag;
public:
    Complex() { real = 0; imag = 0; }
        // 无参构造函数
    Complex(double r, double i) { real = r; imag = i; }
        // 构造函数的重载
    operator double() { return hypot(real, imag); }
        // Complex 向 double 的类型转换运算符，取复数的模
};

int main()
{
    Complex c(3, 4);
    double d = 2.5 + c;         // 调用类型转换函数，类似于 d = 2.5 + double(c1)
    cout << d << endl;
    return 0;
}
```

2.2.3 类型转换冲突

当类的运算符重载、转换构造函数、类型转换运算符重载的定义不使得语句有二义性时，程序能够正常运行。例如上面的例子中

- 2.2.1 节转换构造函数的例子中 double → Complex 转换构造函数、double + double 标准类型运算和 Complex + Complex 运算符重载同时存在，Complex 与 double 的加法表达式没有二义性；

- 2.2.2 节类型转换运算符的例子中只存在 Complex → double 类型转换运算符和 double + double 标准类型运算，Complex 与 double 的加法表达式也没有二义性，即使存在 double → Complex 转换构造函数也没有；
- 容易类推，若同时存在 Complex → double 类型转换运算符、double + double 标准类型运算和 Complex + Complex 运算符重载，Complex 与 double 的加法表达式也没有二义性。

但是，若同时定义了 double → Complex 转换构造函数、Complex → double 类型转换运算符、double + double 标准类型运算和 Complex + Complex 运算符重载，那 Complex 与 double 的加法表达式就有二义性，程序编译不通过，编译器报错：

```
错误(活动) E0350  有多个运算符 "+" 与这些操作数匹配:
                  内置运算符 "arithmetic + arithmetic"
                  函数 "operator+(Complex c1, Complex c2)" (已声明 所在行数:16)
                      操作数类型为: double + Complex
```

类型转换函数可以简化代码编写，利用隐式自动类型转换可以将原本多个运算符函数合并成一个，但过多地使用隐式的自动类型转换会给程序带来隐患。编写程序时应当尽量使用显式类型转换，少用隐式类型转换。

2.3 多态性与虚函数

2.3.1 虚函数与动态绑定

虚函数是为解决多层继承的同名函数统一调用问题而引入的。在基类与派生类中，如果有同名函数，我们只能通过类名限定符（如写明 deobj.Base::func() 与 deobj.func()）来唯一标识，以调用不同派生层次中的同名函数。若不这样，通过对象访问时将应用同名覆盖规则（1.6.3 节）访问对象所属类的该同名函数，但不便于用数组、指针统一管理；通过基类的指针或引用访问时将应用兼容规则（1.6.4 节），只能访问基类中的该同名函数。例如，给出如下的类：

```
class Point // Base class : Point
{
protected:
    double x, y;
public:
    Point(double a = 0, double b = 0) : x(a), y(b) {}
    void setPoint(double a, double b) { x = a; y = b; }
    double getX() const { return x; }
    double getY() const { return y; }
    friend ostream& operator<<(ostream& os, const Point& p);
};

class Circle : public Point // Derived class : Circle, inherits from Point
{
protected:
    double radius;
public:
    Circle(double a = 0, double b = 0, double r = 0) : Point(a, b), radius(r) {}
    void setRadius(double r) { radius = r; }
    double getRadius() const { return radius; }
    double getArea() const { return 3.1415926 * radius * radius; }
    void printArea() const { cout << "Area of the circle is: " << getArea() << endl; }
```

```
23      friend ostream& operator<<(ostream& os, const Circle& c);
24  };
25
26  class Cylinder : public Circle // Derived class : Cylinder, inherits from Circle
27  {
28  protected:
29      double height;
30  public:
31      Cylinder(double a = 0, double b = 0, double r = 0, double h = 0) : Circle(a, b, r),
            height(h) {}
32      void setHeight(double h) { height = h; }
33      double getHeight() const { return height; }
34      double getVolume() const { return 3.1415926 * radius * radius * height; }
35      void printVolume() const { cout << "Volume of the cylinder is: " << getVolume() <<
            endl; }
36      void printArea() const { cout << "Area of the cylinder is: " << 2 * getArea() + 2 *
            3.1415926 * radius * height << endl; }
37      friend ostream& operator<<(ostream& os, const Cylinder& c);
38  };
```

则下面的主函数

```
40  int main()
41  {
42      Circle* arrMix[5];
43      for (int i = 0; i < 3; i++)
44          arrMix[i] = new Cylinder(0, 0, i, i);
45      for (int i = 3; i < 5; i++)
46          arrMix[i] = new Circle(0, 0, i);
47      for (int i = 0; i < 5; i++)
48          arrMix[i]->printArea();
49      return 0;
50  }
```

将输出

```
Area of the circle is: 0
Area of the circle is: 3.14159
Area of the circle is: 12.5664
Area of the circle is: 28.2743
Area of the circle is: 50.2655
```

也即，由于指针数组定义为基类的指针，根据兼容规则只会调用基类的 printArea() 方法。但为了统一管理，不同类的对象又需要在同一数组之中，这就造成了不便。

而，引入虚函数后，用同一语句「*pobj*→func();」，既能调用派生类的同名函数，也能调用基类的同名函数。在调用前给指针变量 *pobj* 赋以不同类对象，即可调用不同派生层次中的 func() 函数，从而实现动态多态性。虚函数定义的一般格式为：

```
virtual <-虚函数的返回类型-> <-函数名-> { <-函数体-> }
```

在上面的例子中，即将第 22 行和第 36 前分别加 virtual。此时，同样的主函数的输出就变为

```
Area of the cylinder is: 0
Area of the cylinder is: 12.5664
Area of the cylinder is: 50.2655
Area of the circle is: 28.2743
Area of the circle is: 50.2655
```

总的来说，虚函数的作用就是允许在派生类中重新定义与基类同名的函数，且可以**通过基类指针或引用来访问**基类和派生类中的同名函数。同一类族中不同类的对象，对同一虚函数的调用将作出不同的响应。

注 2.4. 虚函数的声明注意

（1）在基类中，用 virtual 声明成员函数为虚函数，这样就可以在派生类中重新定义此函数，为它赋予新的功能；

（2）在派生类中重新定义此函数时，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，只能根据派生类的需要重新定义函数体；

（3）在类外定义虚函数时，不必再加 virtual；基类已声明的虚函数在派生类中的同名函数都自动成为虚函数，因此在派生类重新声明该虚函数时，virtual 可加可不加，但建议仍加上以提高可读性。

函数名确定调用的具体对象的过程称为**关联**或**绑定**，其即是把一个函数名与一个类对象捆绑在一起的过程。函数重载和通过对象名调用的虚函数，在编译时即可确定其调用的函数属于哪一个类，因此称为**静态关联**或**编译时关联**；对应地，通过基类指针或引用调用的虚函数在编译时并不知道指针或引用的内容对应于哪一个函数体，必须要到运行时才能确定具体调用的函数版本，所以这种函数行为的绑定称为**动态绑定**，又叫**运行时绑定**。

注 2.5. 虚函数与重载函数的区别

（1）重载函数函数名称相同，参数不同；重载函数，是在作用域相同的区域里定义的相同名字的不同函数；虚函数函数原型完全一致，体现在基类和派生类的类层次结构中；

（2）重载函数可以是成员函数或友员函数或一般函数，而虚函数只能是成员函数；

（3）调用重载函数以所传递参数序列的差别作为调用不同函数的依据，虚函数则根据对象的不同来调用不同类的虚函数；

（4）重载函数在编译时表现出多态性，是静态联编（早期绑定）；而虚函数在运行时表现出多态性，是动态联编（晚期绑定），动态联编是 C++ 的精髓。

一类特殊的情况是，当用基类指针指向派生类对象时，对指针使用 delete 运算符撤销对象，调用的是基类的析构函数。因此，如果有用基类指针统一管理派生类对象的需求，**最好将基类析构函数声明为虚函数**。

2.3.2 纯虚函数与抽象类

基类中定义虚函数可能不是基类本身需求，而是派生类的需要。如上面例子中，基类 Point 中没有面积，但在其派生类 Circle 和 Cylinder 中都需要 printArea() 函数。结构性更强的做法是在最原始的基类 Point 中声明一个没有函数体、只有函数名的虚函数（而非向上例中在中间派生类 Circle 中声明虚函数），这个虚函数称为**纯虚函数**。纯虚函数声明的一般格式为：

```
virtual <-函数类型-> <-函数名-> (<-参数表列->) = 0;
```

纯虚函数只有函数名而不具备函数功能，不能被调用。其作用是在基类中为其派生类保留一个函数名，以便派生类根据需要对它进行定义。声明一个纯虚函数的唯一目的就是**在后续派生中实现多态性**。

因为纯虚函数不能被调用，包含纯虚函数的类无法建立对象，因而称为**抽象类**（abstract class）。声明一个抽象类并不是用来定义对象，而只是作为基类去建立派生类。用户在抽象类的基础上根据需要赋予纯虚函数功能，定义出功能各异的派生类，再用这些派生类去建立对象。

注 2.6. 抽象类的使用

面向对象的程序设计，其层次结构的顶部是抽象类，甚至有几层都是抽象类。如果在抽象类所派生出的新类中对所有纯虚函数进行了定义，被赋予了功能。则该派生类就成为可定义对象的**具体类**（concrete class），否则仍为抽象类。

抽象类不能定义对象，但**可定义指向抽象类数据的指针变量**。当派生类成为具体类后，就可以用该指针指向派生类对象，然后通过指针调用虚函数，实现多态性。

抽象类不能作为参数、函数或显式转换等类型。

2.3.3 多态性

我们希望实现，程序向不同对象发送同一个消息（调用同名函数），不同对象在接收时会产生不同行为、实现不同功能，以此提高代码的利用效率。这种特性称为程序的**多态性**。多态能对不同对象使用同样的接口，既提高程序维护的可靠性，又使得不同对象的内部设计与外部使用接口分离，类的设计者与类的使用者可以分工协作。

多态性按照其主体可以分为

- **重载多态**：函数或运算符重载，依据参数的不同实现；
- **强制多态**：数据或对象的类型强制转换；
- **包含多态**：研究类族中定义同名成员函数的多态行为，主要通过虚函数来实现；
- **参数多态**：类模板实例化时的多态性，即实例化后的各个类都具有相同的操作。

按照其实现可以分为

- **静态多态性**：编译时多态性，例如函数重载、运算符重载、强制类型转换等；
- **动态多态性**：运行时多态性，程序运行过程中动态确定操作所针对的对象，主要通过虚函数实现。

抽象基类体现了本类族中各类的共性，把本类族中共有的成员函数集中在抽象基类中声明（尽管对其功能完全没有设计），作为本类族的公共接口。从抽象基类派生出的多个类具有共同的接口。多态的真正威力在于，程序员可以事先设计一些使用其它模块的代码，对模块的实现细节却可以一无所知。这样，一个模块的修改乃至于功能的添加都不会影响到其它模块的代码。

3 工具

3.1 输入输出流

流是从一个对象到另一个对象的数据流动的字节序列。内存中为每个流开辟了一个内存缓冲区，用来存放流中的数据，即流与内存缓冲区相对应。

3.1.1 C++ 的 I/O 流结构

C++ 的输入、输出（I/O）分为这样三种类型：

- **标准 I/O** 对系统指定的标准设备进行输入输出操作；
- **文件 I/O** 以磁盘文件为对象进行输入输出操作，即从磁盘文件输入、输出数据到磁盘文件；
- **字串 I/O** 对内存中指定空间（通常为一个字符数组）进行输入和输出。

相应地，就有三种**输入输出流**：

- **标准流** 用于程序与输入输出设备之间的数据交互；
- **文件流** 用于程序与磁盘文件之间的数据交互；
- **字串流** 用于程序与内存字符数组之间的数据交互。

C++ 编译系统会对数据类型进行严格检查，凡是类型不正确数据都不能通过编译，确保**类型安全**；C++ 的输入输出命令不仅可以用来输入、输出标准类型的数据，也可输入输出用户自定义类型的数据，即具有**可扩展性**。

C++ 的 I/O 流被定义为不同的**流类**（stream class）。`ios` 是抽象基类，派生出 `istream` 类支持 I 操作、`ostream` 类支持 O 操作，这两个派生类派生出 `iostream` 类支持 I/O 操作。`istream`、`ostream` 类分别派生出 `ifstream` 和 `ofstream` 类支持文件操作，`ifstream` 支持对文件的 I 操作，`ofstream` 支持对文件的 O 操作。用流类定义的对象称为**流对象**，`cout` 和 `cin` 是 iostream 类的全局对象。

3.1.2 标准 I/O 流操作

标准 I/O 流实现程序中对象与标准 I/O 设备（对象）之间的数据交换。进行标准输出操作时，程序对象数据先预存在缓存区中，直到满或者遇到 `endl`，缓冲区的全部数据自动送到屏幕或磁盘文件；输入时，键盘键入的数据先预存到缓冲区，当按回车键，才读入到程序中的对象。

A) >> 与 << 运算符 `istream`、`ostream` 类分别以**成员函数**的形式重载了运算符 >>、<< 作为标准类型的流提取、插入运算符，其格式均为

```
istream & operator >> (<-标准类型->);
ostream & operator << (<-标准类型->);
```

而对于自定义类型数据（自定义类的变量），则需要重载这两个运算符为自定义类的**友元函数**，即如 2.1.3 节 A) 所述，形式为：

```
istream & operator >> (istream &, <-自定义类名-> &);
ostream & operator << (ostream &, <-自定义类名-> &);
```

B) 输出格式控制 `cout` ≪ 输出数据时，系统会判断数据类型，根据其类型选择调用与之匹配的运算符重载函数，将数据插入到对应的内存缓冲区中；当向 cout 流插入一个 `endl` 时，终止一行字符串并刷新缓冲区；插入 `ends` 时，终止字符串并插入空格符。

cout 流的输出格式可有下面两种方式控制：

- **使用控制符控制输出格式**：头文件 `<iomanip>` 中定义了输出流控制符如表 4 所示。

表 4: 输入输出流的控制符

控制符	该控制符的作用	
`dec`	基数	使用十进制
`hex`		使用十六进制
`oct`		使用八进制
`setbase(`*n*`)`		设置整数的基数为 *n*（可取值为 `8` | `10` | `16`）
`setfill(`*c*`)`	对齐	设置「填充字符」为 *c*
`setw(`*n*`)`		固定下一输出项宽度为 *n* 位字符，不足用「填充字符」填充
`setprecision(`*n*`)`	小数表示	设置实数的精度为 *n* 位： 以一般十进制小数形式输出时，*n* 代表有效数字； 以固定小数位数（`fixed`）形式或科学计数（`scientific`）形式输出时，*n* 代表小数位数
`setiosflags(ios::`*FLAG*`)`	设置格式	启用 *FLAG* 格式状态，可选的 *FLAG* 见表 5
`resetiosflags(ios::`*FLAG*`)`	清除格式	终止已设置的 *FLAG* 格式状态

表 5: 可选的 *FLAG* 格式状态标志

FLAG	格式状态的含义	
`dec`	基数	使用十进制
`hex`		使用十六进制
`oct`		使用八进制
`left`	对齐	左对齐
`right`		右对齐
`internal`		符号位左对齐，数值右对齐，中间由「填充字符」填充
`showbase`	记号	强制输出基数标志 `0` 和 `0x`
`showpoint`		强制输出浮点数的小数点和尾数 0
`uppercase`		科学计数形式标志 `e` 和十六进制形式标志 `x` 换为大写
`showpos`		输出正数时输出「`+`」号
`fixed`	小数形式	设置浮点数以固定小数位数形式表示
`scientific`		设置浮点数以科学计数形式表示
`skiows`	流控制	忽略前导空格
`unitbuf`		每次输出之后刷新所有流
`stdio`		每次输出之后刷新 stdout 流和 stderr 流

- **使用流对象的成员函数控制输出格式**：控制输出格式的流对象成员函数定义在流类中，与对应的控制符有相同的作用，如表 6 所示。

表 6: 控制输出格式的流对象成员函数

流成员函数	对应的控制符	该函数的作用	
`fill(c)`	`setfill(c)`	对齐	设置「填充字符」为 *c*
`width(n)`	`setw(n)`		固定下一输出项宽度为 *n* 位字符，不足用「填充字符」填充
`precision(n)`	`setprecision(n)`	小数表示	设置实数的精度为 *n* 位： 以一般十进制小数形式输出时，*n* 代表有效数字； 以固定小数位数（`fixed`）形式或科学计数（`scientific`）形式输出时，*n* 代表小数位数
`setf(ios::FLAG)`	`setiosflags(ios::FLAG)`	设置格式	启用 *FLAG* 格式状态，*FLAG* 见表 5
`unsetf(ios::FLAG)`	`resetiosflags(ios::FLAG)`	清除格式	终止已设置的 *FLAG* 格式状态

C) **标准错误流** 调试程序时，往往不希望程序运行时的出错信息被送到文件，而要求在显示器上及时输出，为此 `ostream` 类中还构建了 2 个流对象 `cerr` 和 `clog`，称为**标准错误流对象**，其作用是向标准错误设备输出有关出错信息，被指定与显示器关联。`cerr`、`clog` 流中的信息是用户根据需要指定的。

这两个流的区别在于，cerr 流不经过缓冲区，直接向显示器上输出有关信息，而 clog 流中的信息存放在缓冲区中，缓冲区满后或遇 `endl` 时才向显示器输出。

D) **标准输入输出函数** 标准输入、输出类的对象，还可以使用以下这些类成员函数进行输入输出。

`put()` `ostream` 类提供的专用于**输出单个字符**的成员函数，使用如

```
cout.put('a');
```

即输出字符 `a`。`put()` 函数的参数可是字符或字符的 ASCII 代码，也可是一个整型表达式。

`get()` `istream` 类提供的成员函数，其有三个重载，即三种调用形式：

- 无参数调用，如

```
char ch = cin.get();
```

从输入流中读取一个字符，返回值是读入字符。若遇到输入流中的文件结束符，则返回文件结束标志 `E0F`。

- 单参数调用，如

```
cin.get(ch);
```

从输入流中读取一个字符，赋给其参数字符变量 *ch*。如果读取成功则函数返回非 0 值，如失败（遇文件结束符）则函数返回 0。

- 三参数调用，其调用形式为

```
cin.get(<-字符数组或指针->, <-字符个数->, <-终止符->);
```

从输入流中读取 <-字符个数-> − 1 个字符，如果在读取 <-字符个数-> − 1 个字符之前遇到指定的终止字符，则提前结束读取。如果读取成功则函数返回非 0 值，如失败（遇文件结束符）则函数返回 0 值。

getline() istream 类还提供了成员函数 getline()，其调用形式及作用与上面 get() 的三参数形式相同。

注 3.1. getline(char *, int, char) 和 get(char *, int, char) 的区别

cin.getline() 和 cin.get() 的功能基本相同，其区别在于因遇到终止符停止读取时，cin.get() 虽不会将终止符保存在数组中，但**仍然保留在输入流中**，因此除非使用 cin.ignore() 等将终止符从输入流中删除，否则紧接着的第 2 个 cin.get() 操作结果为空；而 cin.getline() 要**从输入流中删除终止符**（即读取它并删除），也不将它保存在数组中。

getline() 在读入 <-字符个数-> − 1 字符后会清除缓存，但遇到结束符提前结束不清空。

eof() 从输入流读取数据，若到达文件末或 Ẑ（Ctrl+Z），返回非零值，否则返回 0。

peek() 返回指针指向的当前字符，指针仍停留在当前位置，并不后移。如果要访问的字符是结束符，则返回 EOF（值为 −1 的全局宏）。

ignore() 定义为

```
istream & ignore(std::streamsize _Count = 1i64, int _Metadelim = EOF)
```

逐个跳过输入流中的若干字符，直至刚刚跳过指定的 <-终止符-> *_Metadelim* 或跳过的字符数达到 <-字符个数-> *_Count*。

3.1.3 文件 I/O 流操作

文件指存储在外部介质上数据集合，是**外存的数据管理单位**。文件流是以文件作为 I/O 对象的数据流。每一个文件流都有一个内存缓冲区与之对应。C++ 定义了 3 种用于文件 I/O 操作的**文件类**：

- ifstream 类：istream 类的派生类，支持文件输入；
- ofstream 类，ostream 类的派生类，支持文件输出；
- fstream 类，iostream 类的派生类，支持文件输入输出。

A) 文件的打开与关闭 文件的**打开**是指为**文件流对象**和指定的文件建立关联，以便使文件流流向指定的磁盘文件，并指定文件工作方式的操作。打开文件的方式有以下 2 种：

- 调用**文件流对象的成员函数 open()**，其一般形式为：

```
<-文件流对象->.open(<-文件名->, <-输入输出方式->);
```

例如：

```
ofstream outfile;
outfile.open("f1.dat", ios::out);
```

- 文件流类在声明时，定义了带参数的构造函数，其中包含了打开磁盘文件的功能。因此在定义文件流对象时**调用构造函数的这一重载**来实现「三个功能」，例如：

```
ofstream outfile("f1.dat", ios::out)
```

其中，「输入输出方式」可取值见表 7，且可用「位或」运算符「|」进行组合，如 `outfile.open("f2.dat", ios::app | ios::nocreate);`。

表 7: 可选的文件输入输出方式

输入输出方式标记	输入输出方式说明	
ios::in	工作方向	以输入方式打开
ios::out		以输出方式打开；如有同名文件，则将原有内容全部清除（默认缺省值）
ios::app		以输出方式打开；如有同名文件，则新写入的数据添加在文件末尾
ios::ate		打开一个已有的文件；文件指针指向文件末尾
ios::trunc		打开一个文件；如有同名文件，则将原有内容全部清除（已经指定 out 而未指定 app、in、ate 等时默认）
ios::binary	二进制	以二进制格式打开文件（不指定时默认以 ASCII 码格式打开）
ios::nocreate	不新建	打开一个已有的文件；若文件不存在则打开失败
ios::noreplace	不替换	打开一个新文件；若存在同名文件则打开失败

如果打开操作失败，open() 函数的返回值为 0（后来版本返回值为 void，需要使用 is-open() 来判断）；如果是用调用构造函数的方式打开文件，则流对象的值为 0。这样，就可以先判断文件是否打开，从而判断异常，如：

```
if(!outfile.open("f2.dat", ios::app))
{
    cout << "open error";
    exit(1);
}
```

关闭是解除该磁盘文件与文件流的关联，原来设置的工作方式也失效，这样，就不能再通过文件流对该文件进行 I/O。关闭文件使用成员函数 close()，如 `outfile.close();`。

B) ASCII 文件的读写 ASCII 文件中每个字节均以 ASCII 代码形式存储，即一个字节存放一个字符。程序可从 ASCII 文件中读出或写入若干个字符。

由于文件流类是 istream 类或 ostream 类的派生类，ASCII 文件关联到文件流对象后，可以直接用流插入运算符「<<」和流提取运算符「>>」写入和读出标准类型的数据到文件，也可以用文件流的 put()、get()、getline() 等成员函数进行字符的读写。

注 3.2. .log 日志文件

在软件调试时，一个好技巧是将软件运行中的一些信息同时写到「日志文件」和屏幕上，以便程序员查看信息。写入.log 日志文件使用文件输出流对象，输出到屏幕使用标准输出流对象，而两者输出内容完全一样，于是可以用文件/屏幕输出流 teestream 类对象来同时输出以简化代码。如：

```
std::ofstream outfile("debug.log", ios::out);
outflie << __FILE__ << ": " << __LINE__ << "\t" << "Variable x = " << x;
std::cout << __FILE__ << ": " << __LINE__ << "\t" << "Variable x = " << x;
```

可以改为：

```
std::ofstream outfile("debug.log", ios::out);
teestream tee(std::cout, outfile);
tee << __FILE__ << ": " << __LINE__ << "\t" << "Variable x = " << x;
```

编译器给出了许多宏定义和宏命令，来向程序中引入代码的信息。

- __FILE__：字符串常量，表示当前源文件名；
- __LINE__：十进制整型常量，代表当前程序所在代码行的行号；
- #line <-行号-> <-文件名(可选)->：将行号和文件名更改为指定的行号和文件名；
- __func__ 和 __FUNCTION__：字符串常量，表示当前函数的函数名；
- __DATE__：字符串常量，表示日期，形式为「Mmm dd yyyy」；
- __TIME__：字符串常量，表示时间，形式为「hh:mm:ss」。

C) 二级制文件的读写 **二进制文件**是内存中数据存储形式不加转换地传送到磁盘保存的文件，又称为内存数据的**映像文件**或字节文件。打开二进制文件时，需用 ios::binary 指定为以二进制形式打开。二进制文件除了可以作为输入文件或输出文件外，还可是既能输入又能输出的文件，即关联到 fstream 类流对象。

用成员函数 read 和 write 读写二进制文件的基本格式为

```
istream & read(char * <-字符指针buffer->, int <-字节数len->);
ostream & write(const char * <-字符指针buffer->, int <-字节数len->);
```

其中字符指针 buffer 指向内存中一段存储空间，即，保存文件中 *len* 字节的数据到从指针所指向位置开始的内存空间，或读出从指针所指向位置开始 *len* 字节的内存空间数据到文件。

D) 文件指针操作 每一个打开的文件都有一个**文件指针**，指向文件中当前应进行读写的位置。文件流对象中与文件指针有关的成员函数如表 8 所示。

利用成员函数移动文件指针，可以随机地访问到文件中任一位置的数据。

3.1.4 字串 I/O 流操作

字符串流是以内存中用户定义的字符数组（字符串）为对象的 I/O 流，也称为**内存流**，其也有相应的缓冲区。如果向字符数组存入数据，当流缓冲区满或遇换行符时，缓冲区中的字符一起存入字符数组。如果是从字符

表 8: 与文件指针有关的文件流对象成员函数

流成员函数	该函数的作用	
gcount()	计数	返回此前最后一次输入所读入的字节数
tellg()	输入调整	返回输入文件指针的当前位置
seekg(*n*)		移动输入文件指针到**绝对**位置 *n* 字节处
seekg(*n*, ios::cur)		移动输入文件指针到**相对当前**位置 *n* 字节处
seekg(*n*, ios::end)		移动输入文件指针到**相对文件尾**位置 *n* 字节处
tellp()	输出调整	返回输出文件指针的当前位置
seekp(*n*)		移动输出文件指针到**绝对**位置 *n* 字节处
seekp(*n*, ios::cur)		移动输出文件指针到**相对当前**位置 *n* 字节处
seekp(*n*, ios::end)		移动输出文件指针到**相对文件尾**位置 *n* 字节处

数组读数据，先将字符数组中的数据送到流缓冲区，然后从缓冲区中提取数据赋给有关变量。C++ 定义了 3 种用于字符串 I/O 操作的**字符串流类**：

- istrstream 类：istream 类的派生类，支持字串输入；
- ostrstream 类，ostream 类的派生类，支持字串输出；
- strstream 类，iostream 类的派生类，支持字串输入输出。

这些类都定义在头文件 <strstream> 中。

与文件打开类似，字符串流也需要自己建立字符串流对象，并与指定的字符数组关联。ostrstream 类构造函数声明为

```
ostrstream::ostrstream(char * <-字符数组->, int <-缓冲区大小->, int <-输入输出方式-> =
    ios::out);
```

strstream 类构造函数与此相同。istrstream 类有两个带参构造函数的重载，声明分别为：

```
istrstream::istrstream(char * <-字符数组->);
istrstream::istrstream(char * <-字符数组->, int <-读取长度->);
```

若使用后者，则只讲字符数组中前 <-读取长度-> 个字节的内容放入输入字符串流。

在将其他类型数据转换到字符串流中时，应特别注意如何分隔两个相邻的数据，以便于从字符串流中读出。

注 3.3. 字符串流的使用范围

（1）标准流不能保存数据，文件流保存数据需要建立、打开文件，而字符数组中的内容可以随时用 ASCII 字符输出，比外存文件使用方便，存取速度更快；

（2）字符数组的生命周期与其所在的模块（如主函数）相同，该模块的生命周期结束后，字符数组也不存在了，因此字符串流只适合短期保存数据。

3.2 异常处理

异常是指程序在运行时超出了程序员预计的某些特殊情况，不在正常的情况之列，如文件打开失败、内存分配失败、外部模块调用失败、非法指针、非法运算（除数为 0）、数组访问越界、函数输入输出参数值超出预期

范围、未初始化的变量使用、算法逻辑错误等。程序除了处理正常情况，还要对异常情况进行及时甄别，正确处理，以防引发更大的错误。

造成异常的原因有两种：

- 一是因为我们的程序自身存在错误，无法完成所要求的工作。程序的错误包括编译时错误和运行时错误两种，即使编译通过的程序，仍然可能存在严重的错误和漏洞，导致运行时错误。
- 二是因为我们的程序所调用的外部模块发生错误，致使我们的程序无法继续正常运行。每一个程序都需要依赖外部模块提供的输入输出、函数运算等系统功能，我们的程序在调用这些外部函数时，可能会遇到错误。

C++ 对异常采用**结构化处理**的方式，其基本思想是发现与处理分离、逐级上报：

- 发现与处理分离机制。使底层的函数专门用于解决实际任务，而不必再承担处理异常的任务，以减轻底层函数的负担而把处理异常的任务上移到某一层去处理，可提高效率。
- 逐级上报机制。如果在执行一个函数过程中出现异常，发出一个信息给它上一级（即调用它的函数），上级捕捉到信息后进行处理。如果上一级函数也不能处理，就再传给其上一级。如此逐级上报。如果到最高一级（操作系统级）还无法处理，最后调用 terminate() 终止程序。

C++ 异常处理的 **try-catch 结构**为

```
try
{
    // 经过一系列检查语句发现异常
    throw <-异常信息->;
}
catch(<-异常类型->)
{
    // 进行异常处理
}
```

其中，throw 抛出的<-异常信息->是一个表达式，throw 抛出这个表达式后其所在函数即结束执行，先在本函数中找与之**数据类型匹配**的 catch，如无 try-catch 结构或找不到匹配的 catch，则转到最近的上级 try-catch 结构。如最终找不到匹配的 catch 块，则系统会调用函数 terminate() 来终止程序运行。

注 3.4. try-catch 结构的使用

（1）try-catch 结构具有整体性。catch 块须紧跟 try 块，二者之间不能插入其他语句。但在一个 try-catch 结构中，可只有 try 块而无 catch 块，或一个 try 块后跟多个 catch 块。

（2）catch 语句检测的只是信息类型，因此 catch 的参数可只标明类型，如 catch(double)。如没有指定类型，而是删节号「…」，则表示可捕捉任何类型异常信息。

（3）如果在 catch 的参数中写入变量，如 catch(double *b*)，则可实现 throw *a*; 中 *a* 到 *b* 值传递。

（4）throw 可不包括表达式，表示这一级不处理此异常，请上级处理，调用栈向上传播。

下面是一个例子。

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

int main()
```

```
5  {
6      void f1();
7      try
8      { f1(); }
9      catch (double)
10     { cout << "Caught double exception" << endl; }
11     cout << "End 0" << endl;
12     return 0;
13 }
14
15 void f1()
16 {
17     void f2();
18     try
19     { f2(); }
20     catch (char)
21     { cout << "Caught char exception" << endl; }
22     cout << "End 1" << endl;
23 }
24
25 void f2()
26 {
27     void f3();
28     try
29     { f3(); }
30     catch (int)
31     { cout << "Caught int exception" << endl; }
32     cout << "End 2" << endl;
33 }
34
35 void f3()
36 {
37     double a = 0.;
38     try
39     { throw a; }
40     catch (float)
41     { cout << "Caught float exception" << endl; }
42     cout << "End 3" << endl;
43 }
```

在第 39 行抛出的 double 异常最终被第 9 行的 catch 块捕获，程序输出为

```
Caught double exception
End 0
```

如果让第 40 行捕获 double 异常（41 行不修改），则程序输出变为

```
Caught float exception
End 3
End 2
End 1
End 0
```

注 3.5. 抛出异常的预声明

为了函数调用时的阅读性，建议声明函数时列出可能抛出的异常类型，缺省是指可能抛出任何类型的异常信息，如：

```
double triangle(double, double, double) throw(double);
double triangle(double, double, double) throw(int, float, char);
```

其中，异常信息可是标准或自定义类型数据。如果想声明一个不抛出异常的函数，则可写为：

```
double triangle(double, double, double) throw();
```

异常指定必须同时出现在函数声明和函数定义的首行中。否则，编译系统将报告「类型不匹配」。

除了抛出标准类型的异常之外，还可以抛出自定义类的异常，即 throw 一个自定义类对象。这样，理论上就可以满足任意的错误分类。

还可以对第三方函数的抛出异常进行封装，以适应本程序的异常规则。

3.3 命名空间

C++ 引入可由用户命名的作用域，用来解决程序中同名冲突问题。在不同的文件（编译单元）、函数、复合语句、类作用域等作用域中，可定义相同名字变量，互不干扰。**命名空间**（namespace）就是由应用程序来命名的内存区域。通过命名空间机制，可以把一些全局实体分别放在各自空间中，从而来实现与其他全局同名实体区分。定义命名空间的基本格式为

```
namespace <-命名空间名->
{
    // 空间中的所有全局实体，可以嵌套命名空间
}
```

命名空间中的变量通过 <-命名空间名->::<-变量名-> 的格式调用。

利用 using 语句，可以声明本文件中某个变量名均指这一命名空间中的这一变量，如：

```
using nsp::Student;
Student stu("Huang Yongfeng", 18);  // 等价于 nsp::Student stu(”Huang Yongfeng”, 18);
```

若直接使用整个空间，则可用 using namespace 语句，如：

```
using namespace nsp;
Student stu("Huang Yongfeng", 18);  // 等价于 nsp::Student stu(”Huang Yongfeng”, 18);
```

注 3.6. 关于 using namespace std

std 是标准 C++ 库的命名空间，标准 C++ 头文件中函数、类、对象和类模板都是在命名空间 std 中定义的。因此，在程序中用到 C++ 标准库时，需要使用 std 作为限定。最简便的方式自然是在文件开头加入 using namespace std; 语句，这样在 std 中定义和声明的所有标识符在本文件中都可以作为全

局量来使用。但是，这时应当绝对保证在程序中不出现与命名空间 std 的成员同名的标识符。

3.4 模板和容器

3.4.1 函数模板和类模板

模板是一种**使用「数据类型」作为参数**来产生一系列函数或类的机制，采用「类型参数」来完成不同的功能，可让用户得到类或函数声明的一种通用模式，使得类中的某些数据成员或者成员函数的参数、返回值取得不同类型。模板是 C++ 支持多态性的一种工具，体现出 C++ 泛化（通用）编程思想，方便了更大规模的软件开发；减少了程序员编写代码的工作量。

函数模板的本质是建立一个通用函数，其函数的数据类型和形参类型不具体指定，用虚拟类型表示。定义一个函数模板的格式为：

```
template <typename/class <-虚拟类型名->>
// 通用函数的定义
```

函数调用时，系统会根据实参类型来取代模板中虚拟类型，从而得到实现不同功能的**模板函数**，这称为**函数模板的实例化**，例如：

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

template <class T>
void outputArray(const T* P_array, const int count)
{
    for (int i = 0; i < count; i++)
        cout << P_array[i] << " ";
    cout << endl;
}

int main()
{
    const int aCount = 8, bCount = 8, cCount = 20;
    int aArray[aCount] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    double bArray[bCount] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8 };
    char cArray[cCount] = "Welcome to see you";
    cout << "a Array contains:" << endl;
    outputArray(aArray, aCount);
    cout << "b Array contains : "<<endl;
    outputArray(bArray, bCount);
    cout << "c Array contains:" << endl;
    outputArray(cArray, cCount);
    return 0;
}
```

凡是多个函数体相同的函数都可合并为函数模板表示。

注 3.7. 函数模板的使用

（1）函数模板的说明（定义）必须写在全局作用域，不能说明为类的成员函数，更不能嵌套在函数中。
（2）由于模板类型实参的任意性，函数模板不具有隐式类型转换的作用。
（3）模板函数也可以重载。在匹配调用函数时，首先匹配类型完全相同的重载函数，然后匹配类型适合的函数模板重载。只要有类型适合的函数模板重载，就不会进行隐式类型转换去调用类型不完全相同的重载函数。

类似于函数模板，如果若干个类的功能相同，仅仅是数据类型不同，则可以声明一个通用的**类模板**，其格式为：

```
template <typename/class <-虚拟类型名->>
class <-类名->
{
    // 通用成员的定义
};
```

类模板的成员函数也可以在类模板之外定义，一般格式为：

```
template <typename/class <-虚拟类型名->>
<-函数类型-> <-类模板名-><<-虚拟类型名->> :: <-函数名->(<-形参表->)
{
    // 函数体
}
```

类模板可使类中的某些数据成员、成员函数的参数或返回值能取任意类型。如果说类是对象的抽象，对象是类的实例，则**类模板是类的抽象，类是类模板的实例**。类模板也称为「参数化类」。

使用类模板时，在类模板名之后用尖括号指定类型名，程序编译时系统就会用指定类型来取代模板中虚拟类型，从而得到不同的**模板类**，这称为**类模板的实例化**。

类模板也可以进行派生。用类模板派生出新的类模板的格式为

```
template <typename/class <-基类模板的虚拟类型名->>
class <-派生类模板名-> : <-派生方式-> <-基类模板名-><<-基类模板的虚拟类型名->>
{
    // 派生类模板定义
};
```

用模板类派生出新类的格式为

```
template <typename/class <-基类模板的虚拟类型名->>
class <-派生类名-> : <-派生方式-> <-基类模板名-><<-指定的实际类型名->>
{
    // 派生类定义
};
```

*3.4.2 STL 与容器

标准模板库（Standard Template Library，STL）采用模板和可重用组件，实现了通用数据结构以及处理这些数据的算法。STL 是一套程序库，也是软件重用技术发展史上突破，提出了一种泛型（通用）程序设计模式。STL 引入了一下这些概念：

- **容器** 常用数据结构的模板化，可表示 Vector、List、Map 等各种数据结构对象，每个容器表现为一个类模版。
- **算法** sort、search、copy 等常用操作函数的通用模板，适用于不同类型的数据，以迭代器作为函数参数，可灵活地处理不同长度数据集合。
- **迭代器** 算法和容器间的「桥梁」，是一种泛型指针，保存它所操作的特定容器状态。迭代器指向容器中某位置，用运算符函数「++」或「--」前后移动，用「*←迭代器→」表示指向数据。
- **函数对象** 行为类似函数，可作为演化算法的某种策略（policy），传递算法操作的特定规则。

容器分为三类：

- **顺序容器** 可变长动态数组 vector、双端队列 deque、双向链表 list。其特点是元素在容器中的位置同元素的值无关，即容器是不排序的，便于扩展。
- **关联容器** 集合 set、multiset、图 map、multimap。特点是元素是排序的。默认关联容器中的元素是从小到大排序；具有很好的查找性能。
- **容器适配器** 在两类容器的基础上屏蔽一部分功能，突出或增加另一部分功能，实现了栈 stack、队列 queue、优先级队列 priority_queue 这 3 种适配器。