湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课 程 名 称	算法设计与分析，计算机图形学，数字图像处理，计算机图形图像技术
章节、专题	必备的C++基础
教学目标及基 本 要 求	熟悉一些必备的C++基础知识，掌握C++标准模板库的使用。
教 学 重 点	类和对象，模板，C++ STL
教 学 难 点	类和对象，模板
教学内容与时 间 分 配	（1）C++对C的几种过程性扩充（1.2课时） （2）类和对象（1.3课时） （3）继承机制（1.5课时#） （4）多态性与虚函数（1.5课时#） （5）模板（0.7课时） （6）C++ STL的使用（1.3课时） 共计5课时（8课时#）。
习　　题	第A.7节（练习题）。

附录A　必备的C++基础

本附录简要介绍一些必备的C++基础知识，演示C++标准模板库中常用算法函数和数据结构的使用。

- C++对C的几种过程性扩充
- 类和对象
- 模板
- C++ STL使用举例

A.1 C++对C的几种过程性扩充

在本章中，C++默认为C++ 14，C++ 14以后的新增特性会特意指明。

A.1.1 输入和输出

1. C++版的“Hello”程序

以“Hello”程序为例演示如何使用C++标准模版库完成最基本的输入和输出。

```
#include<iostream> // cin, cout, endl;
#include<string> // string
using namespace std;
int main() // C++标准要求返回int值
{   std::cout << "Input your name." << std::endl; // 显示“Input your name.”
    std::string name; // 保存姓名
    std::cin >> name; // 假设输入“John”
    cout << "Hello, " << name << "!" << endl; // 显示“Hello, John!”
    return 0; // 可省略
}
```

```
#include<iostream> // cin, cout, endl;
#include<string> // string
using namespace std;
int main() // C++标准要求返回int值
{   std::cout << "Input your name." << std::endl; // 显
    std::string name; // 保存姓名
    std::cin >> name; // 假设输入“John”
    cout << "Hello, " << name << "!" << endl; // 显
    return 0; // 可省略
}
```

2. 输入流和输出流

C++通常使用标准输入流对象std::cin和标准输出流对象std::cout进行输入和输出。

（1）输入流。std::cin必须与输入运算符“>>”配套使用。例如，“std::cin >> x;”将输入数据转换为变量x的类型并存入x。运算符“>>”允许连续输入数据，如“std::cin >> x >> y >> z;”。

（2）输出流。std::cout必须与输出运算符“<<”配套使用。例如，“std::cout << y;”将变量y的值显示在屏幕上。使用运算符“<<”进行输出操作时，可以把多个不同类型的数据组合在一条语句中，也可以输出表达式的值。例如，“std::cout << "a + b= " << 2.5 + 3.5 << std::endl;”将依次输出字符串“a + b=”和表达式2.5 + 3.5的值并换行，结果为“a + b= 6”。

【注】std::cin、std::cout和std::endl等在文件iostream中定义。在使用“using namespace std;”后，std::cin、std::cout和std::endl等可以简写为cin、cout和endl等。

A.1.2 变量的声明和作用域

1. 变量声明和类型转换

- C++允许在源程序的任何合适位置声明变量，只要符合“先定义，后使用”的规定即可。
- C++允许定义运行期定长数组。例如，“int n = 9; double X[n], Y[n] = {1, 2, 3};”定义了运行期定长数组X和Y。注意，初始化时初始值数量不能超过数组长度。
- 在C++中，结构（结构体）名、联合（共用体）名、枚举名等都是类型名，在定义变量时，类型名不需要以struct、union或enum等关键字开头。
- C++可以使用decltype(表达式)指定变量类型，也允许使用auto由编译器推断变量类型。例如，对于int变量i，“decltype(&i) p;”相当于“int *p;”，“auto q = &i;”相当于“int *q = &i;”。
- C++增加了一种类似于函数调用的类型转换格式。例如，“x = double(i)”和“p = (double *)(&i)”。

2. 引用

引用相当于对象的一个新名字，通常使用形如“类型 &引用名 = 对象”的形式声明。使用这种形式声明的引用允许通过引用修改相应对象，但是只能引用左值。带有const限定的引用通常称为常值引用（const引用），通常使用形如“const 类型 &引用名 = 对象”的形式声明。常值引用允许引用左值和右值，但是不能通过引用修改相应对象。注意，在声明引用时必须指明该引用表示的对象。

下述示例程序演示了引用的声明和使用。

```
#include<iostream>
using namespace std;
int main()
{   int v = 10; // 变量定义
    int &r = v; // 引用说明, r是对变量v的引用, 不是新变量
    r = 20; // 通过引用修改相应对象
    cout << (r == v) << endl; // 1 (r与v值相同)
    cout << (&r == &v) << endl; // 1 (r与v地址相同)
    const int &cv = 50; // 常值引用, 不能通过引用修改相应对象
    // 不允许 cv = 10;
}
```

3. 函数的参数与返回值

（1）引用参数。C++支持传引用的参数传递方式。这种方式在形参声明时，在参数前加上引用说明符。在调用函数时，传递实参的引用。函数通过引用存取实参，可以修改实参的值。例如，在下述例子中，调用Swap(x, y)会交换变量x和y的值。

```
#include<iostream>
using namespace std;
void Swap(int &x, int &y)
{   int temp = x;
    x = y;
    y = temp;
}
int main()
{   int x = 1, y = 2;
    Swap(x, y);
    cout << x << ", " << y << endl; // 2, 1
}
```

```
void Swap(int *x, int *y)
{   int temp = (*x);
    (*x) = (*y);
    (*y)= temp;
}
int main()
{   int x = 1, y = 2;
    Swap(&x, &y);
    // ...
}
```

【注】通常，引用参数只能接受左值。如果希望在调用时可以根据需要引用左值或右值，则可以考虑使用常值引用参数。

（2）返回引用。可以使用引用类型作为函数的返回值类型，返回某个对象的引用。当然，必须保证该对象在函数返回后依然存在。采用这种方法可以将函数调用放在赋值运算符的左边。

```
#include<iostream>
using namespace std;
int &at(int X[], int i)
{   return X[i - 1]; // 便于使用从1开始的下标
}
int main()
{   int X[10];
    at(X, 1) = 5; // 等价于X[0]=5
    cout << X[0] << endl; // 5
}
```

（3）返回值类型的后置与推导。C++允许使用auto作为返回值类型的占位符，在函数头部的末尾使用“->返回值类型”指定函数的返回值类型，这就是返回值类型后置。

如果函数体中只有一个return语句，或所有return语句中的值类型都相同，则可以省略后置的返回值类型，将return语句中的值类型用作返回值类型，这就是返回值类型推导。

```
#include<iostream>
using namespace std;
auto Max(int x, double y) -> decltype(x + y) // 返回值类型后置(或double)
{   if(x > y) return x;
    else return y;
}
auto Min(double x, double y) // 返回值类型推导
{   if(x < y) return x;
    else return y;
}
int main()
{   cout << Max(1.1, 2.2) << ", " << Min(1.1, 2.2) << endl; // 2.2, 1.1
}
```

4. 结构化的绑定声明

C++ 17允许对变量和引用使用结构化的绑定声明，使用方法如下例。

```
#include<iostream>
using namespace std;
struct Test { int u, v; };
int main()
{   Test o = { 10, 15 }; // 变量定义
    auto [u, v] = o; // 等价于auto u = o.u, v = o.v
    // u与o.u, v与o.v值相同
    cout << (u == o.u and v == o.v) << endl; // 1
    // u与o.u, v与o.v地址不同
    cout << (&u != &o.u and &v != &o.v) << endl; // 1
    auto &[r, s] = o; // 等价于auto &r = o.u, &s = o.v
    // r与o.u, s与o.v值相同
    cout << (r == o.u and s == o.v) << endl; // 1
    // r与o.u, s与o.v地址相同
    cout << (&r == &o.u and &s == &o.v) << endl; // 1
}
```

【注】C++为一些运算符提供了关键字表示，常用的如表A-1所示。

表A-1　运算符的关键字表示

运算符	关键字
!	not
&&	and
||	or
~	compl
&	bitand
\| | bitor |

运算符	关键字
^	xor
!=	not_eq
&=	and_eq
|=	or_eq
^=	xor_eq

5. 变量的作用域

在变量的作用域方面，C++引入了名字空间的机制，可以使用名字空间解决名字重复等问题。

（1）名字空间的定义。用形如“namespace X { int n = 2; }”的方式定义名字空间，其中，“{ }”中可以定义任何变量、类型和函数等。

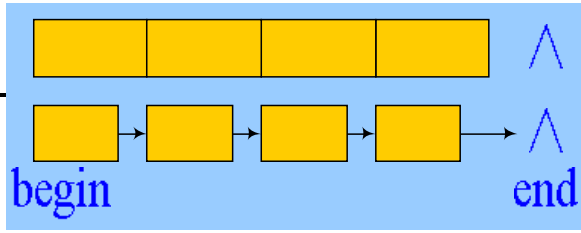
（2）名字空间的引用。名字空间中的名字可以通过下列两种方式引用。

- 通过域解析符引用。名字空间中的名字可以通过域解析符“::”引用（域解析符也可称为域运算符）。例如，cout对象属于std名字空间，可以通过std::cout引用。
- 通过using指示引用。名字空间中的名字也可以通过“using namespace *;”引用。例如，在使用“using namespace std;”命令后，也可以通过cout引用std::cout。

（3）全局变量的引用。如果在当前作用域中有全局变量与局部变量同名的情形，默认使用局部变量。如果需要使用全局变量，则可以在全局变量的名字前面加上域解析符，例如，::X表示引用的是全局变量X。

（4）举例说明。下述例子演示了变量的作用域和名字空间的使用。

```
#include<iostream>
using namespace std;
int m = 0, n = 1; // 全局变量
namespace X { int n = 2; } // 名字空间
int main()
{   int n = 3; // 局部变量
    cout << "global m: " << m << endl; // global m: 0
    cout << "local n: " << n << endl; // local n: 3
    cout << "global n: " << ::n << endl; // global n: 1
    cout << "X::n: " << X::n << endl; // X::n: 2
}
```



A.1.3 基于范围的for循环

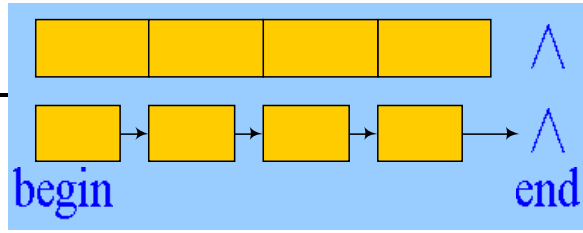
C++支持基于范围的for循环。在使用基于范围的for循环处理一个序列时，该循环对序列中的每个元素自动迭代一次。基于范围的for循环以统一、简洁的方式遍历序列中的每个元素，可以避免越界访问，使用起来比较方便。

【注】这里说的序列对象是指初始值列表、数组、字符串和容器等可以获得起始位置和结束位置，且可以按照某种顺序依次访问每个元素的对象。其中，初始值列表是用大括号{}界定、用逗号分隔的一组初始值。

1. 基本语法

for(循环变量声明 : 范围表达式) 循环语句

- 循环变量声明：声明循环变量，变量类型是范围表达式中的序列元素的类型，或对该类型的引用。通常使用auto自动推断。
- 范围表达式：表示某个序列对象的任何表达式，该循环对序列中的每个元素迭代一次。
- 循环语句：任何语句，通常是复合语句，构成循环体。



2. 举例说明

下面举例说明这种循环的具体用法。

```
#include<iostream>
using namespace std;
int main()
{   int n = 5;
    double X[n] = {0, 1, 2, 3, 4};
    for(auto e : X) e *= 2; // e是各元素的值
    for(int i = 0; i < n; ++i) // 使用经典for显示
        cout << X[i] << ' ';
    cout << endl; // 0 1 2 3 4
    for(auto &e : X) e *= 2; // e是各元素的引用
    for(const auto &e : X) // 使用范围式for显示, e是各元素的常值引用
        cout << e << ' ';
    cout << endl; // 0 2 4 6 8
    for(const auto &e : {1, 2, 3, 4, 5}) // 遍历初始值列表
        cout << e << ' ';
    cout << endl; // 1 2 3 4 5
}
```

A.1.4　动态存储分配

1. 运算符new

运算符new可用于动态存储分配。该运算符返回所分配空间的首地址。运算符new的使用格式如下。

- new 对象类型{初始值}
- new 对象类型[对象个数]{初始值}

第一种格式用于建立单个对象（“{}”可换成“()”），第二种格式用于建立对象数组。“{初始值}”可以省略，使用默认初始值。

2. 运算符delete

动态分配的存储空间不再需要时应该及时释放。在C++中，用运算符delete来释放new运算符分配的空间。delete命令的使用格式如下。

- delete 被释放对象的地址
- delete []被释放对象的首地址

第一种格式用于释放单个对象，第二种格式用于释放对象数组。

【注】两种格式的new和delete必须对应使用，不可混用。否则，很可能得不到期望的结果。

3. 一维数组

为了在运行时创建一个大小可动态变化的一维Type类型数组，可以先将数组名声明为一个Type类型的指针，然后用new运算符为该数组动态分配存储空间。

```
#include<iostream>
using namespace std;
void Print(int X[], int n); // 辅助程序，显示数组的前n个元素
int main()
{   int *X = new int, n = 3;
    *X = 10;
    int *Y = new int{20}; // 或new int(20)
    cout << *X << ' ' << *Y << endl; // 10 20
    int *Z = new int[n];
    fill_n(Z, n, 5); // Z的每个元素都是5
    Print(Z, n), cout << endl; // 5 5 5
    int *W = new int[n] {1, 2, 3};
    Print(W, n), cout << endl; // 1 2 3
    delete X, delete Y, delete[]Z, delete[]W;
}
```

```
void Print(int X[], int n)
{   if(n <= 0) return; // 没有元素
    cout << X[0]; // 首元素
    for(int i = 1; i < n; ++i)
        cout << ' ' << X[i]; // 后续每个元素前面加上空白
}
```

A.1.5　函数重载

1. 什么是函数重载

在C++中，两个或者两个以上的函数可以使用相同的函数名。但是，这些函数的参数列表不同，即这些函数的形参个数、顺序或者类型不同，这就是函数重载。被重载的函数称为重载函数。在调用这些函数时，编译器会根据实际参数列表（即实参的个数、顺序和类型）来确定应该调用哪个重载函数。

注意，如果两个函数的函数名和参数列表都相同，只有返回值类型不同，则不能重载。

2. 举例

假设需要两个Max函数，其中一个返回两个int类型对象中的较大者，另一个返回两个double类型对象中的较大者。

```
#include<iostream>
using namespace std;
int Max(int x, int y)
{   return x > y ? x : y;
}
double Max(double x, double y)
{   return x > y ? x : y;
}
int main()
{   int i = Max(1, 2);
    double d = Max(1.1, 2.2);
    cout << i << ", " << d << endl; // 2, 2.2
    // Max(11, 2.2)歧义
}
```

3. 默认参数

C++允许在说明函数原型时为参数列表右侧的一个或多个参数指定默认值。例如，有一函数原型说明为“void init(int x, int y = 5, int z = 10);”，则y和z的默认值分别为5和10。

在进行函数调用时，编译器按照从左向右的顺序将实参与形参结合。如果没有指定足够的实参，则编译器按顺序用函数原型中的默认值来补足缺少的实参。例如，“init(0, 25)”相当于“init(0, 25, 10)”，“init(0)”相当于“init(0, 5, 10)”。

注意，所有默认参数都必须出现在非默认参数的右边。函数声明和函数调用都必须遵守这项规定。

4. 默认参数与函数重载

默认参数与函数重载不能冲突。例如，如果有2个函数原型说明“void init(int x, int y = 5, int z = 10);”和“int init(int x);”，则调用“init(0)”时不知道应该调用哪个函数，编译器会报错。

A.2 类和对象

类是一种组织数据和函数的自定义数据类型。数据称为这个类的成员变量，函数称为这个类的成员函数。一个类可以有多个对象。

A.2.1 类的定义

1. 类的组成部分

C++的类由4个部分组成。

- 类名。
- 数据成员，也称成员变量。
- 函数成员，也称成员函数。
- 访问级别。

2. 类成员的访问级别

对类成员的访问有3种不同的级别：公有（public）、私有（private）和保护（protected）级别。其中，在public域中声明的成员可以在程序中直接访问，在private域中声明的成员只能由此类的成员函数和友元函数访问。

在C++中，结构体（struct）与类（class）的区别就是结构体中的默认访问级别是public，类中的默认访问级别是private。

3. 示例

（1）类的定义。下列Position类（保存在文件Position_i.h中）描述二维整数坐标，成员函数GetX()用于返回x坐标，GetY()用于返回y坐标。

```
#pragma once // 只编译一次
class Position
{   int x = 0, y = 0; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
    friend bool Equal(const Position &, const Position &); // 友元
    Position &operator=(const Position &); // 运算符重载(成员)
    // 运算符重载(常规, 友元)
    friend bool operator==(const Position &, const Position &);
};
```

（2）相关函数的定义。这里先列出相关函数的定义，然后在后续部分逐一介绍。其中，友元函数见第A.2.6节，运算符重载见第A.2.7节。

```
Position::Position(int x, int y): x(x), y(y) {} // 构造函数
// 拷贝构造函数
Position::Position(const Position &q): Position(q.x, q.y) {}
Position::~Position() {} // 析构函数
```

```
int Position::GetX() const { return x; } // 返回x坐标
int Position::GetY() const { return y; } // 返回y坐标
```

```
bool Equal(const Position &p, const Position &q) // 友元
{   return p.x == q.x and p.y == q.y;
}
```

```
Position &Position::operator=(const Position &p) // 运算符重载(成员)
{   x = p.x, y = p.y;
    return *this;
}
```

```
bool operator==(const Position &p, const Position &q)
{   // 运算符重载(常规, 友元)
    return p.x == q.x and p.y == q.y;
}
```

```
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函
    int GetY() const; // y坐标
    friend bool Equal(const Position &, con
    Position &operator=(const Position &);
    // 运算符重载(常规, 友元)
    friend bool operator==(const Position &
```

A.2.2 成员函数的定义

在类中声明成员变量的通常形式与C语言的struct成员说明基本一致，常用的扩充之处是C++允许为成员变量指定默认值。例如，在类Position的定义中，可将成员变量声明修改为“int x = 0, y = 0;”为成员变量*x*和*y*指定默认值。

【注】成员变量不能使用auto声明。

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
```

1. 声明

在类中声明成员函数的通常形式为

返回值类型 成员函数名(参数列表) [限定];

其中，限定是可选的部分。这里只说明一下最常用的const限定。如果一个成员函数的参数列表后带有const限定，则表示不允许在该成员函数的函数体中修改当前对象的任何数据成员。

下列程序片段给出了类Position的成员函数GetX和GetY的声明。

```
int GetX() const; // x坐标(不允许该函数修改数据成员)
int GetY() const; // y坐标
```

相当于下列 C 风格声明。

```
int GetX(const Position *this);
```

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
```

2. 定义

在类的外部定义成员函数的通常形式为

返回值类型 类名::成员函数名(参数列表) [限定] { 函数体 }

一般将类的声明放在头文件中，而将类的实现放在源程序文件中。当然，也可以在声明成员函数的同时给出函数体。在定义类模板时通常会使用同时给出声明和函数体的方式。

下列程序给出了类Position的成员函数GetX和GetY的定义。

```
int Position::GetX() const { return x; } // 返回x坐标
int Position::GetY() const { return y; } // 返回y坐标
```

实际上，“return x;”是“return this->x;”的简写。在成员函数内部，this指向调用该成员函数的对象，因此，可用“*this”引用相应对象，可用“this->成员名”引用相应对象的指定成员。而且，在无歧义时可以省略“this->”。

相当于下列C风格定义

```
int GetX(const Position *this)
{   return this->x;
}
```

```
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
```

A.2.3 构造函数与析构函数

1. 构造函数

在定义一个对象时，将自动调用构造函数，初始化该对象的数据成员。构造函数名与类名相同，可以有一个或多个参数，在定义对象时传递实际参数值。例如，类Position中的声明“Position(int, int);”和“Position(const Position &);”为类Position声明了两种构造函数。

声明构造函数时，必须注意下述三点。

（1）构造函数通常声明为公有成员（私有构造函数只能在类的内部使用）。

（2）构造函数不可以有返回值也不得指明返回值类型（包括void）。

（3）构造函数的各种重载形式（含构造函数的默认参数）之间都不能有冲突。例如，在类Position中，声明“Position();”与“Position(int = 0, int = 0);”就有冲突。

```
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
```

2. 默认构造函数

默认构造函数是不带参数的构造函数。如果没有显式定义任何构造函数，编译器会生成一个默认构造函数，使得对象的每个数据成员都使用默认值，否则，必须显式定义允许不带参数调用的构造函数。例如，类Position中的声明“Position(int = 0, int = 0);”为类Position显式定义了允许不带参数调用的构造函数。

【注】可以使用形如“Position() = default;”的声明形式显式规定由编译器生成默认的构造函数。

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
```

3. 拷贝构造函数

拷贝构造函数是参数为同类型对象引用的构造函数，用于在函数中创建某个对象的副本，如类Position中的“Position(const Position &);”。如果没有显式定义拷贝构造函数，编译器会生成一个默认的拷贝构造函数，使得当前对象每个数据成员的值都和参数对象对应数据成员相同。

【注】可以使用形如“Position(const Position &) = default;”的声明形式显式规定由编译器生成默认的拷贝构造函数。

4. 析构函数

当对象被释放时，将自动调用析构函数。析构函数的函数名由“~”字符和类名构成，参数列表只能是空的参数列表，如类Position中的“~Position();”。

析构函数主要用于在含有指针的数据成员中自动释放动态数据成员。如果没有动态数据成员，则析构函数通常为空（即使用空的函数体）。空析构函数的说明和定义都可以省略（必须同时省略）。

```
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
```

5. 构造函数和析构函数的定义

（1）定义构造函数的通常形式。通常使用下述形式定义构造函数。

构造函数名(参数列表): 初始化列表 { 函数体 }

其中，初始化列表中的各项之间用“,”分隔，通常用于指定部分成员的初始值，形式通常为“成员名(参数值列表)”或“构造函数名(参数值列表)”。

【注】在初始化列表中只能使用已经定义好的能够初始化本类部分成员的构造函数。

（2）定义析构函数的通常形式。通常使用下述形式定义析构函数。

析构函数名() { 函数体 }

（3）示例。下列程序给出了类Position的构造函数和析构函数的定义。

```
Position::Position(int x, int y): x(x), y(y) {} // 构造函数
Position::Position(const Position &q): Position(q.x, q.y) {} // 拷贝构造函数
Position::~Position() {} // 析构函数
```

【注】这里定义的拷贝构造函数与默认的拷贝函数完成同样的工作，不需要显式定义。同样，析构函数也可以省略。当然，必须同时省略说明和定义。

```
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造
    ~Position(); // 析构函数
```

A.2.4　类的对象

1. 对象的声明

类对象的声明与创建类似于变量的声明与创建。对一个对象成员进行访问或调用可采用直接选择（.）或间接选择（->）实现。

下列代码说明了如何声明类Position的对象，以及如何调用其成员函数。

```
#include<iostream>
#include"Position_i.h"
using namespace std;
int main()
{   Position X{1, 2}; // 构造函数, {}可换成()
    Position *Y = new Position{3, 4}; // 构造函数, {}可换成()
    Position Z{}; // 构造函数, 使用默认参数值, {}可省略, 不可换成()
    cout << X.GetX() << ", " << X.GetY() << endl; // 1, 2 (成员)
    cout << Y->GetX() << ", " << Y->GetY() << endl; // 3, 4
    cout << Z.GetX() << ", " << Z.GetY() << endl; // 0, 0
    cout << Equal(Z, X) << endl; // 0 (友元)
    Z = X; // 运算符重载(成员)
    cout << (Z == X) << endl; // 1, 运算符重载(常规函数, 友元)
}
```

2. 对象参数

（1）值参数。传递给函数的实参是一个对象，在函数中创建该对象的副本。在创建副本时调用拷贝构造函数，在函数调用结束时调用副本的析构函数。

（2）引用参数。采用引用方式传递对象，在函数中不创建该对象的副本，因而也不需要撤销副本。引用参数可能改变引用传递的对象。

通常，一个对象作为值参数传递时，为了节省传递时的资源开销，一般声明为常值引用，即参数类型为const Type &。当然，这种声明不允许在函数体内部修改相应参数。

A.2.5 友元函数

在类声明中使用关键字friend声明的函数称为友元函数。友元函数不是类的函数成员，但友元函数能引用类的私有成员和保护成员。友元函数可以是一个常规函数，也可以是另一个类的成员函数，可以在类声明中的任意区域声明。下列程序定义了类Position的友元函数Equal()。

```
bool Equal(const Position &p, const Position &q) // 友元
{   return p.x == q.x and p.y == q.y;
}
```

注意，必须在类中声明该友元函数。声明形式为

```
friend bool Equal(const Position &, const Position &); // 友元
```

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
    friend bool Equal(const Position &, const Position &); // 友
    Position &operator=(const Position &); // 运算符重载(成员
    friend bool operator==(const Position &, const Position &);
```

A.2.6 运算符重载

1. 运算符重载的含义

运算符重载的目标是使得程序更加直观，可以通过创建运算符函数实现。用关键字operator加上运算符来表示函数名的函数称为运算符函数。例如，两个坐标相等可定义成如下函数。

```
bool Equal(const Position &, const Position &);
```

但人们习惯用“==”来表示相等，这可用运算符重载来表示。

```
bool operator==(const Position &, const Position &);
```

运算符重载与普通函数定义的不同之处是，运算符重载的函数名规定为“operator运算符”，且参数个数必须满足相应运算符的要求。

运算符与普通函数在调用时的不同之处是，普通函数的参数出现在圆括号内，而运算符的参数出现在运算符的左、右两侧。

```
int GetY() const; // y坐标
friend bool Equal(const Position &, const Position &); // 友
Position &operator=(const Position &); // 运算符重载(成员
// 运算符重载(常规, 友元)
friend bool operator==(const Position &, const Position &);
```

```
cout << X.GetX() << ", " << X.GetY() << endl; // 1, 2 (成员)
cout << Y->GetX() << ", " << Y->GetY() << endl; // 3, 4
cout << Z.GetX() << ", " << Z.GetY() << endl; // 0, 0
cout << Equal(Z, X) << endl; // 0 (友元)
Z = X; // 运算符重载(成员)
cout << (Z == X) << endl; // 1, 运算符重载(常规函数, 友元)
```

2. 运算符重载为常规函数

如果运算符被重载为常规函数，那么一元运算符重载为只有一个参数的函数，二元运算符重载为有两个参数的函数。例如，两个坐标相等的运算符“==”可重载为

```
bool operator==(const Position &, const Position &);
```

下列程序实现了对类Position的运算符“==”的重载。

```
// 运算符重载(常规, 友元)
bool operator==(const Position &p, const Position &q)
{   return p.x == q.x and p.y == q.y;
}
```

注意，该运算符重载的实现引用了对象的私有成员或保护成员，必须在类中将该运算符重载声明为类的友元函数，声明形式为

```
friend bool operator==(const Position &, const Position &);
```

经重载运算符“==”后，即可用运算符“==”比较两个Position对象是否相等。

```
int GetY() const; // y坐标
friend bool Equal(const Position &, const Position &); //
Position &operator=(const Position &); // 运算符重载(
friend bool operator==(const Position &, const Position &
```

```
cout << Z.GetX() << ", " << Z.GetY() << endl; // 0, 0
cout << Equal(Z, X) << endl; // 0 (友元)
Z = X; // 运算符重载(成员)
cout << (Z == X) << endl; // 1, 运算符重载(常规函数, 友元)
```

3. 运算符重载为成员函数

如果运算符重载为类的成员函数，那么一元运算符没有参数，二元运算符只有一个右侧参数。例如，坐标复制的运算符“=”可重载如下。

```
Position &operator=(const Position &);
```

下列程序实现了对类Position的运算符“=”的重载。

```
Position &Position::operator=(const Position &p) // 运算符重载
{   x = p.x, y = p.y;
    return *this;
}
```

在重载运算符“=”后，就可以使用运算符“=”将一个Position对象的值规定为另一个Position对象的值。

注意，必须在类的public域中声明该成员函数。声明形式为

```
Position &operator=(const Position &); // 运算符重载
```

```
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
    friend bool Equal(const Position &, const Position &); // 友
    Position &operator=(const Position &); // 运算符重载(成员
    friend bool operator==(const Position &, const Position &);
};
```

```
cout << Y->GetX() << ", " << Y->GetY() << endl; // 3, 4
cout << Z.GetX() << ", " << Z.GetY() << endl; // 0, 0
cout << Equal(Z, X) << endl; // 0 (友元)
Z = X; // 运算符重载(成员)
cout << (Z == X) << endl; // 1, 运算符重载(常规函数, 友元)
```

4. 几点说明

（1）不能重载作用域运算符“::”、成员运算符“.”、指向成员运算符“.*”、sizeof运算符和三目条件运算符“?:”。

（2）类型转换运算符、成员运算符“->”、指向成员运算符“->*”、函数调用运算符“()”、下标运算符“[]”和赋值运算符“=”只能重载为成员函数。

（3）如果没有显式定义任何赋值运算符函数，编译器会生成一个默认的赋值运算符函数，使得当前对象每个数据成员的值都和参数对象对应数据成员相同。否则，必须显示定义赋值运算符函数的所有需要使用的重载形式。

【注】可以使用形如“Position &operator=(const Position &) = default;”的声明形式显式规定由编译器生成默认的赋值运算符函数。

5. 允许重载的常用运算符♫

如表A-2所示。这里列出的运算符是按分类顺序排列的，其中加上“*”号的重载方式表示通常使用的重载方式。

表A-2　允许重载的常用运算符

运算符	原始含义	使用示例	重载方式
++, --	自增，自减	i++; ++j;	成员*
[]	下标	array[4] = 2	成员
->	通过指针访问成员	ptr->mem = 34	成员
*, &	间接访问，地址	u = *p; q = &v;	成员*
+, -	正，负	x = -i	成员*
*, /, %	积，商，模(余数)	m = 5 % 3	常规*
+, -	和，差	j = 4 - 2	常规*
<<, >>	左移位，右移位	x = 33 << 1	常规*
<, <=	小于，小于或等于	if(i < 42) ++i;	常规*
>, >=	大于，大于或等于	if(i > 42) --i;	常规*
==, !=	等于，不等于	if(i != 42) j = 42;	常规*
~	位反	y = ~x	成员*
&, ^, |	位与，位异或，位或	y = x & 7	常规*

运算符	原始含义	使用示例	重载方式
!	逻辑非	if(!done) ++i;	成员*
&&, ||	逻辑与, 逻辑或	if(x > 0 && x < n) y = 0;	常规*
=	赋值	y = x	成员
*=, /=, %=	积(商, 模)赋值	z %= 3	成员*
+=, -=	和赋值, 差赋值	x -= 3	成员*
<<=, >>=	左移赋值, 右移赋值	x <<= 2	成员*
&=, ^=, |=	位与(异或, 或)赋值	x |= 2	成员*
()	函数调用	w = max(x, y)	成员
type(), (type)	类型转换	i = int(3.5); j = (int)13.5;	成员
,	逗号	i = 0, j = 1, k=2	常规*

A.2.7# 静态成员

在类定义中，关键字static声明未绑定到类实例的成员。

1. 静态数据成员

（1）基本语法。

- static 数据类型 静态数据成员名;
- inline static 数据类型 静态数据成员名[= 初始值];

（2）定义与初始化。

- 使用前一种语法声明一个静态数据成员时不能同时初始化，必须在类外定义和初始化。在类外定义和初始化静态数据成员的一般格式为

  数据类型 类名::静态数据成员名[= 初始值];

- 后一种语法需C++17支持，用于在类的内部定义和初始化静态数据成员。

2. 静态函数成员

（1）基本语法。

- static 返回值类型 静态函数成员名(参数表);
- static 返回值类型 静态函数成员名(参数表){ 函数体 }

（2）定义。

- 使用前一种语法声明一个静态函数成员时必须在类外定义该成员函数。在类的外部定义静态函数成员的一般格式为

  返回值类型 类名::静态函数成员名(参数表){ 函数体 }

- 后一种语法用于在类的内部定义一个静态函数成员，即在声明静态函数成员的同时给出函数体。

3. 静态成员的引用

可以使用两种形式引用类T的静态成员m，一种是使用限定名T::m，另一种是使用成员访问表达式E.m或P->m，其中E和P分别是T类型和T*类型的表达式。在同一个类的范围内，可以直接使用静态成员的名字，不需要限定。

静态成员服从类成员的访问规则（私有，受保护，公有）。

4. 几点注意

（1）静态数据成员。可以将静态数据成员理解为带有类名的服从类成员访问规则的全局变量。

- 静态数据成员不与任何对象关联，即使没有定义该类的对象，它们也存在。
- 在整个程序中只有一个静态数据成员实例。

（2）静态函数成员。可以将静态函数成员理解为带有类名的服从类成员访问规则的常规函数。

- 静态函数成员不与该类的任何对象关联，调用时没有this指针。
- 静态函数成员不能是virtual、const或volatile的。

（3）在类外定义静态数据成员或静态函数成员时不能带上static，但必须带上类名。

5. 示例

下述示例演示静态成员的定义和使用。

```
#include<iostream>
using namespace std;
class Foo
{   int _id_; // 对象编号(普通数据成员)
    inline static int _next_id_ = 0; // 下一对象编号(私有静态数据成员)
public:
    Foo(): _id_(_next_id_) // 构造函数
    {   ++_next_id_; // 下一对象编号, 类的内部无需限定
    }
    int id() const // 对象编号(普通成员函数, const的)
    {   return _id_;
    }
    static int next_id() // 下一对象编号(公有静态函数成员)
    {   return _next_id_;
    } // 不能是virtual, const或volatile的
};
```

```
int main()
{   Foo X[10];
    for(auto &e : X)
        cout << e.id() << " "; // 调用普通成员函数
    cout << endl; // 0 1 2 3 4 5 6 7 8 9
    cout << Foo::next_id() << ", " << X[0].next_id(); // 调用静态函数成员
    cout << endl; // 10, 10
}
```

A.3# 继承机制

类的继承机制使程序员无需修改已有类，只需在已有类的基础上，通过增加少量代码或修改少量代码的方法得到新的类，从而较好地解决了代码重用的问题。

由已有类产生新类时，通常将已有类称为基类或父类，新类称为派生类或子类。由基类产生出派生类通常称为由基类派生出派生类或派生类继承自基类。此时，派生类包含了基类的特征，同时也加入了自己的新特性。

A.3.1 派生类的声明

这里只讨论单继承的情况，多继承的情况在后面再讨论。

单继承是指派生类只有一个基类，多继承是指派生类有多个基类。

如果没有特别说明，继承通常表示单继承。

1. 声明派生类的基本格式

（1）声明格式。声明一个派生类的基本格式为

```
class 派生类名:[继承方式] 基类名
{   派生类新增的数据成员和成员函数
};
```

（2）几点说明。

- 继承方式指明如何规定基类成员在派生类中的访问级别，包括关键字private、protected、public（分别表示私有、保护、公有继承，默认为private）。
- 在定义派生类时，可以在派生类中说明与基类成员同名的成员。这时，在派生类中可以使用“基类名::成员名”的格式访问基类的同名成员。

2. 基类成员在派生类中的访问级别

基类成员在派生类中的访问级别如表 A-3 所示，可以总结为以下几点。

- 基类的私有成员在派生类中不可见。
- 公有继承不改变访问级别。
- 保护继承将访问级别改为保护的。
- 私有继承将访问级别改为私有的。

表A-3　基类成员在派生类中的访问级别

基类成员	公有继承	保护继承	私有继承
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	不可见	不可见	不可见

3. 派生类的静态成员

派生类和派生类对象在遵循成员访问规则的前提下可以直接使用基类的静态成员，这些静态成员也是派生类的静态成员。

当然，派生类也可以定义新的静态成员。和普通成员一样，在派生类中使用“基类名::成员名”的格式访问基类的同名成员。

4. 举例

下述示例演示了基类成员在派生类中的访问级别。

（1）基类。

```
#include<iostream>
using namespace std;
class Base // 基类
{
public: int publ; // 公有数据成员
protected: int prot; // 保护数据成员
private : int priv; // 私有数据成员
public:
    Base(int x, int y, int z): publ(x), prot(y), priv(z) {}
    void show() const
    {   cout << publ << ", " << prot << ", " << priv;
    }
    ~Base()
    {   cout << "~Base()" << endl;
    }
};
```

（2）公有继承。

```
class PublicDerive: public Base // 派生类,公有继承
{   // publ, prot, priv, Base::Base(), Base::show()来自基类
    // publ公有, prot保护, priv不可见
    int own; // 新增, 私有
public:
    // : publ(x), prot(y), priv(z), own(w)有错
    PublicDerive(int x, int y, int z, int w): Base(x, y, z), own(w) {}
    void show() const
    {   // cout << publ << ", " << prot << ", " << priv有错
        Base::show(); // 基类成员,公有
        cout << ", " << own; // 新增
    }
    ~PublicDerive()
    {   cout << "~PublicDerive(), ";
    }
};
```

基类成员	公有继承	保护继承	私有继承
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	不可见	不可见	不可见

（3）保护继承。

```
class ProtectedDerive: protected Base // 派生类,保护继承
{   // publ保护, prot保护, priv不可见
    int own; // 新增, 私有
public:
    // : publ(x), prot(y), priv(z), own(w)有错
    ProtectedDerive(int x, int y, int z, int w): Base(x, y, z), own(w) {}
    void show() const
    {   // cout << publ << ", " << prot << ", " << priv有错
        Base::show(); // 基类成员, 公有
        cout << ", " << own; // 新增
    }
    ~ProtectedDerive()
    {   cout << "~ProtectedDerive(), ";
    }
};
```

基类成员	公有继承	保护继承	私有继承
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	不可见	不可见	不可见

（4）私有继承。

```
class PrivateDerive: private Base // 派生类,私有继承
{   // publ私有, prot私有, priv不可见
    int own; // 新增, 私有
public:
    // : publ(x), prot(y), priv(z), own(w)有错
    PrivateDerive(int x, int y, int z, int w): Base(x, y, z), own(w) {}
    void show() const
    {   // cout << publ << "," << prot << "," << priv有错
        Base::show(); // 基类成员, 公有
        cout << "," << own; // 新增
    }
    ~PrivateDerive()
    {   cout << "~PrivateDerive(), ";
    }
};
```

基类成员	公有继承	保护继承	私有继承
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	不可见	不可见	不可见

（5）派生类对象的声明与使用。

```
int main()
{   PublicDerive X(1, 2, 3, 4); // 公有,可见
    X.Base::show(), cout << endl; // 公有
    X.show(), cout << endl; // 公有
    ProtectedDerive Y(1, 2, 3, 4); // 公有,可见
    // Y.Base::show()有错(Base::show保护, 不可见)
    ((Base &)Y).show(), cout << endl; // 转换为基类引用, 公有, 可见
    Y.show(), cout << endl; // 公有, 可见
    PrivateDerive Z(1, 2, 3, 4); // 公有, 可见
    // Z.Base::show()有错(Base::show私有, 不可见)
    ((Base &)Z).show(), cout << endl; // 转换为基类引用, 公有, 可见
    Z.show(), cout << endl; // 公有,可见
}
```

```
1, 2, 3                          ~PrivateDerive(), ~Base()
1, 2, 3, 4                       ~ProtectedDerive(), ~Base()
1, 2, 3                          ~PublicDerive(), ~Base()
1, 2, 3, 4
1, 2, 3
1, 2, 3, 4
```

基类成员	公有继承	保护继承	私有继承
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	不可见	不可见	不可见

A.3.2　派生类的构造函数和析构函数

1. 派生类定义构造函数的格式

在 C++中，简单派生定义构造函数的一般格式为

```
派生类名(参数表): 基类名(基类参数), 新增成员的初始化
{   其他初始化工作
}
```

其中，基类构造函数参数表中的参数通常来源于派生类构造函数的参数表，也可以使用常量和表达式。

2. 几点说明

- 基类构造函数不带参数时，派生类不一定需要显式定义构造函数。当基类的构造函数带有参数时，派生类必须显式定义构造函数。
- 可以使用“using 类名::类名;”的方式引入基类的构造函数。
- 如果派生类的基类也是一个派生类，每个派生类只负责其直接基类的构造，依次上溯。

3. 派生类析构函数的工作机制

在派生类中可以根据需要定义自己的析构函数，用于对派生类的新增成员进行清理，基类成员的清理仍然由基类的析构函数负责。

在执行派生类的析构函数时，系统会自动调用基类的析构函数，对基类的成员进行清理。

在派生类中是否需要自行定义析构函数与它所属基类的析构函数无关，它们是各自独立执行的。

A.3.3　赋值兼容规则

1. 赋值兼容规则的含义

通过公有继承，可以将公有派生类对象的值赋给基类对象，也可以在用到基类对象的时候用其公有派生类对象代替，不过只能使用基类成员（虚函数例外），这就是基类和派生类对象之间的赋值兼容规则。

2. 赋值兼容规则的具体表现

这里使用下述示例类演示赋值兼容规则的具体表现。

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

```
struct Base // 基类
{   int val;
    Base(int i): val(i) {}
    void show() const { cout << "Base::val = " << val; }
};
```

```
struct Derive: public Base // 赋值兼容只对公有派生类有效
{   Derive(int i): Base(i) {}
    void show() const { cout << "Derive::val = " << val; }
};
```

```
void setValue(Base &o, int i) // 基类引用
{   o.val = i; // 通过基类引用修改对象
}
```

```
void setValue(Base *p, int i) // 基类指针
{   p->val = i; // 通过指针引用修改对象
}
```

（1）派生类对象可以赋值给基类变量或传递到基类值参数，不过只能对基类对象的数据成员赋值。

```
int main()
{   Base X(10); // 基类对象X
    Derive Y(20); // 派生类对象Y
    X.show(), cout << ", ", Y.show(), cout << endl;
    X = Y; // 不允许Y = X
    X.show(), cout << ", ", Y.show(), cout << endl;
    Y.val = 50; // 修改派生类对象
    X.show(), cout << ", ", Y.show(), cout << endl;
```

```
Base::val = 10, Derive::val = 20
Base::val = 20, Derive::val = 20
Base::val = 20, Derive::val = 50
```

（2）派生类对象可以初始化基类引用或传递到基类引用参数，但是通过该引用只能访问从基类继承的成员，不能访问派生类新增的成员。

```
    setValue(Y, 60); // 通过基类引用修改派生类对象
    Y.show(), cout << endl;
```

```
Derive::val = 60
```

（3）派生类对象的地址可以赋值给基类指针变量或传递到基类指针参数，但是通过该指针只能访问从基类继承的成员，不能访问派生类新增的成员。

```
    setValue(&Y, 70); // 通过基类指针修改派生类对象
    Y.show(), cout << endl;
}
```

```
Derive::val = 70
```

A.3.4 多继承

1. 多继承的声明

（1）多继承声明的基本形式。在多继承中，声明派生类的基本形式为

```
class 派生类: 继承方式1 基类1, ..., 继承方式n 基类n
{  新增成员
};
```

在多继承中，各种继承方式对于基类成员在派生类中的访问属性和规则与单继承相同。

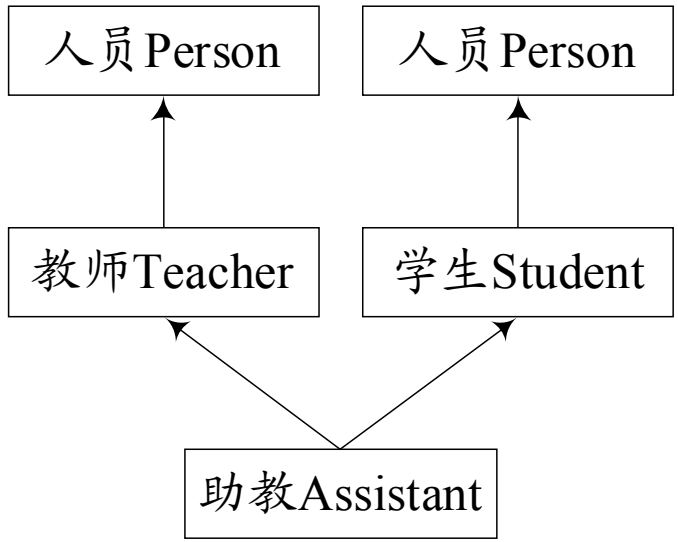
（2）多继承中派生类构造函数的基本形式。多继承构造函数定义的基本形式为

```
派生类(参数表): 基类1构造, …, 基类n构造, 新增成员初始化
{ 其他初始化语句
}
```

2. 多继承举例

派生类 OnJobStudent 从基类 Employee 和 Student 派生，而这两个基类有一个共同的基类 Person。

（1）继承关系。如图 A-1 所示。



图A-1　多继承

（2）共同基类 Person。

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

```
class Person
{   // 人员，共同基类
protected:
    string name; // 姓名
```

```
public:
    Person(const string &name): name(name) {}
```

```
    void show() const
    {   cout << name;
    }
};
```

（3）基类 Teacher。

```
class Teacher: public Person
{   // 员工, 基类
protected:
    string dept; // 工作部门
public:
    Teacher(const string &name, const string &dept):
        Person(name), dept(dept) { }
    void show() const
    {   Person::show();
        cout << ", " << dept << "(Dept)";
    }
};
```

（4）基类 Student。

```
class Student: public Person
{   // 学生, 基类
protected:
    string major; // 主修专业
public:
    Student(const string &name, const string &major):
        Person(name), major(major) { }
    void show() const
    {   Person::show();
        cout << ", " << major << "(Major)";
    }
};
```

（5）派生类 Assistant。

```
class Assistant: public Teacher, public Student
{   // 助教，派生类，多继承
public:
    Assistant(const string &name, const string &dept, const string &major):
        Teacher(name, dept), Student(name, major) { }
    void show() const
    {   Teacher::show();
        cout << ", " << major << "(Major)";
    }
};
```

（6）派生类对象。

```
int main()
{   Teacher X("Ada", "Computers"); // 计算机系教师
    X.show(), cout << endl;
    Student Y("Alice", "Maths"); // 数学专业学生
    Y.show(), cout << endl;
    Assistant Z("Tom", "Computers", "Maths"); // 助教
    Z.show(), cout << endl;
}
```

```
Ada, Computers(Dept)
Alice, Maths(Major)
Tom, Computers(Dept), Maths(Major)
```

A.4# 多态性与虚函数

A.4.1 多态性概述

1. 多态性的含义

在C++程序设计中，多态性是指用一个名字定义不同的函数，这些函数执行不同但又类似的操作，这样就可以用同一个函数名调用不同内容的函数。

2. 多态性的实现

从实现的角度来讲，多态性可以分为编译时多态和运行时多态2类。编译时多态是通过静态联编来实现的。运行时多态是通过动态联编实现的。所谓联编就是把函数名与函数体的程序代码连接（联系）在一起的过程。

3. 静态联编与动态联编

静态联编是在编译阶段完成的联编。静态联编在编译时就能确定调用哪个函数，函数调用速度很快，效率高，但缺乏灵活性。

动态联编是运行阶段完成的联编。动态联编在运行时才能确定调用哪个函数，降低了程序的运行效率，但增强了程序的灵活性。

在 C++中，编译时多态性主要通过函数重载和运算符重载实现。运行时多态性主要通过虚函数实现。

A.4.2 虚函数

虚函数是在基类中用关键字 virtual 说明，可以在派生类中重新定义的成员函数。关键字 virtual 指示 C++编译器，使用引用或指针调用虚函数时，采用动态联编的方式，即在运行时确定调用哪个函数。虚函数定义后，就可以通过基类指针或引用访问基类和派生类中的虚函数。

1. 虚函数的定义

虚函数的定义是在基类中进行的，只需在需要定义为虚函数的成员函数的声明前加上关键字 virtual 即可。

可以在派生类中重新定义虚函数。在派生类中重新定义虚函数时，必须满足以下几点要求。

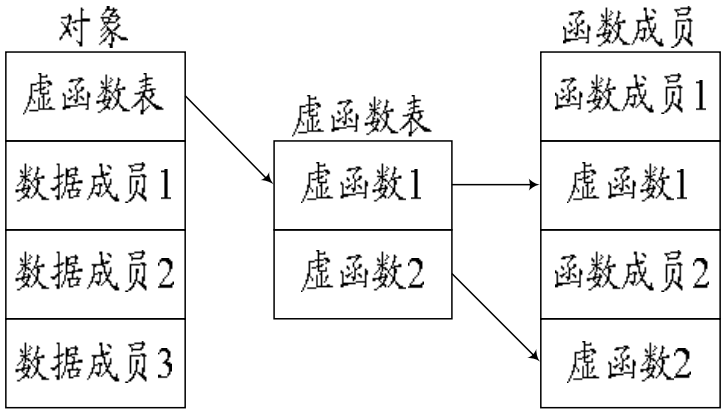
- 该函数与基类的虚函数有相同的名称。
- 该函数与基类的虚函数有相同的参数列表。
- 该函数与基类的虚函数有相同的限定（如 const, volatile 等）。
- 该函数与基类的虚函数有相同的返回值类型或者满足赋值兼容规则的指针、引用型的返回值类型。

2. 虚函数表

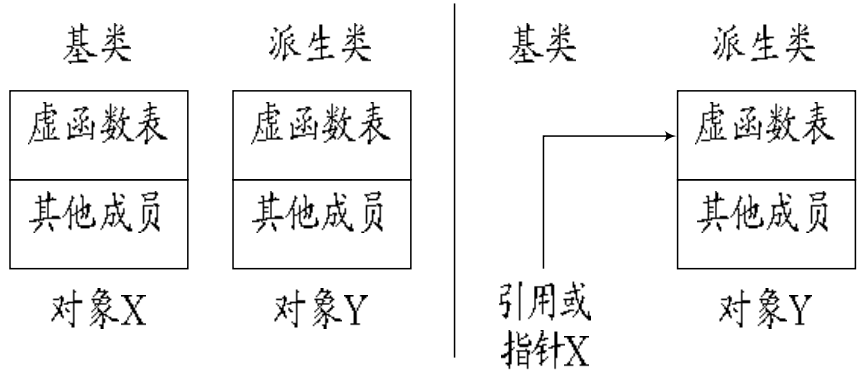
含有虚函数的类对象中有一个虚函数表，在运行时通过虚函数表确定应该调用哪个函数，如图 A-2 所示。

如果使用派生类对象初始化基类变量，则基类变量与派生类对象是两个独立对象，基类变量不能访问派生类对象的虚函数表，如图 A-3 左侧所示。

如果通过基类的引用或指针引用派生类对象，则该引用或指针可以访问派生类对象的虚函数表，如图 A-3 右侧所示。



图A-2　内存布局



图A-3　访问虚函数

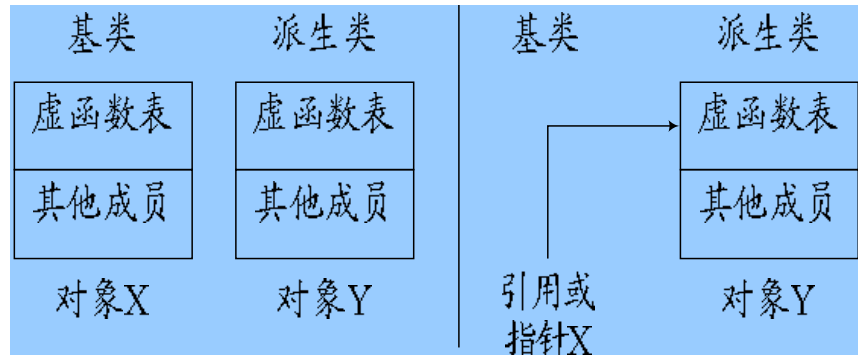
3. 几点说明

（1）关于虚函数声明和定义需要注意以下几点。

- 在类中声明虚函数时需带 virtual，在类外定义虚函数时不能带 virtual。
- 虚函数只能是所在类的对象成员函数，不能是友元函数和静态函数成员。
- 构造函数不能是虚函数，但是析构函数可以是虚函数，而且在有虚函数的类中通常将析构函数定义为虚函数。

（2）关于多态性需要注意以下几点。

- 通过定义虚函数来使用多态性机制时，派生类必须从基类公有派生。
- 只有通过基类的引用或指针访问虚函数才能获得运行时多态。因为只有基类的引用或指针才能引用派生类对象，才能通过派生类对象的虚函数表访问派生类对象的虚函数。如图 A-3 所示。
- 在调用虚函数时，如果在成员函数名前缀上了类名，则该调用使用静态联编，不会使用虚函数表。



（3）关于虚函数重新定义需要注意以下几点。

- 在派生类中重新定义虚函数时，如果只有函数名与基类虚函数相同，而参数列表或限定说明与基类虚函数存在差异，则是普通的成员函数重载，不是虚函数重新定义。
- 如果在派生类中重新定义了基类的虚函数或重载了同名函数，则会隐藏基类中的其他同名函数，所以在派生类中需要重新定义所有必需的同名函数。也可以只定义必须重新定义的同名函数，其余同名函数使用“using 基类名::成员函数名;”的方式从基类引入。
- 在派生类中重新定义虚函数时，关键字 virtual 可以省略，也可以在限定说明后加上 override 关键字，通知编译器检查该定义是否真的是基类虚函数的重新定义。
- 如果在派生类中没有重新定义基类的虚函数，则派生类继承其直接基类的虚函数。一个虚函数无论被继承多少次都会保持虚函数特性。

4. 举例说明

使用基类指针引用派生类对象。

（1）基类。含有一个虚函数和一个重载函数。

```
#include<iostream>
using namespace std;
struct Base
{   virtual void show() const //  虚函数
    {   cout << "Base::show() virtual";
    }
    void show(int i) const //  重载函数
    {   cout << "Base::show(" << i << ")";
    }
};
```

（2）派生类。含有一个虚函数和两个重载函数。

```
struct Derive: public Base // 赋值兼容只对公有派生类有效
{   void show() const override // 虚函数
    {   cout << "Derive::show() virtual";
    }
    void show(int i) const // 重载函数
    {   cout << "Derive::show(" << i << ")";
    }
    void show(const char *s) const // 重载函数
    {   cout << "Derive::show(" << s << ")";
    }
};
```

（3）派生类对象。使用基类指针引用派生类对象。

```
int main()
{   Base X, *p; // 基类对象X，基类指针p
    Derive Y; // 派生类对象Y
    p = &X; // 指向基类对象
    p->show(), cout << endl; // Base::show()
    p = &Y; // 指向派生类对象
    p->show(), cout << endl; // Derive::show()
    p->show(15), cout << endl; // Base::show(int)
    Y.show(15), cout << endl; // Derive::show(int)
    Y.show("Test"), cout << endl; // Derive::show(const char *)
    // 不能使用p->show("Test"), 因为没有Base::show(const char *)
}
```

```
Base::show() virtual
Derive::show() virtual
Base::show(15)
Derive::show(15)
Derive::show(Test)
```

A.4.3 纯虚函数与抽象类

1. 纯虚函数

纯虚函数在基类中只声明而在派生类中实现的虚函数。

（1）声明形式。声明纯虚函数的一般形式为

virtual 返回值类型 函数名(参数表) = 0;

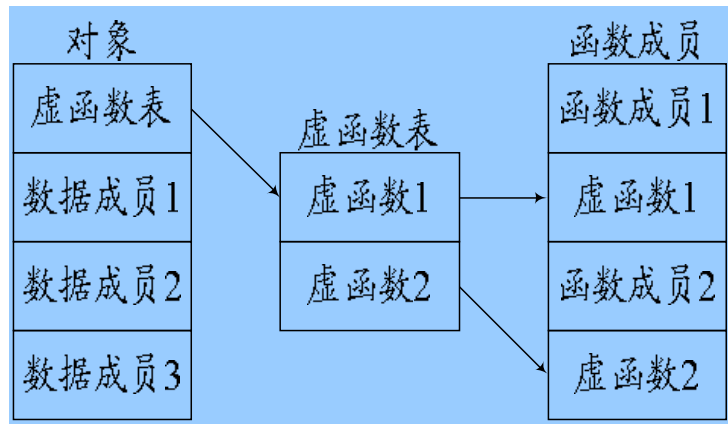
（2）几点说明。

- 纯虚说明符“ = 0”只是指明该虚函数是纯虚函数，不表示返回值为 0。
- 纯虚函数必须在派生类中重新定义，否则仍然是纯虚函数。

2. 抽象类

含有纯虚函数的类称为抽象类，没有纯虚函数的类称为具体类。抽象类表示一般概念（如 Shape 和 Animal 等），可以用作具体类（如 Circle 和 Dog 等）的基类。

注意，抽象类不能用于创建对象，只能用于通过引用或指针引用其派生类（具体类）对象。



3. 应用举例

应用抽象类求圆、圆内接正方形和圆外切正方形的面积。

（1）基类 Shape。含有一个纯虚函数。

```
#include<iostream>
using namespace std;
struct Shape
{   Shape(double x): r(x) { }
    virtual double area() const = 0; // 纯虚函数
protected:
    double r; // 半径
};
```

（2）派生类 Circle、InSquare 和 ExSquare。重新定义了虚函数。

```
struct Circle: public Shape
{   Circle(double x): Shape(x) { }
    double area() const override { return 3.14 * r * r; } // 虚函数
};
```

```
struct InSquare: public Shape
{   InSquare(double x): Shape(x) { }
    double area() const override { return 2 * r * r; } // 虚函数
};
```

```
struct ExSquare: public Shape
{   ExSquare(double x): Shape(x) { }
    double area() const override { return 4 * r * r; } // 虚函数
};
```

（3）多态性。使用基类引用访问派生类对象。

```
void show(const Shape &X, const char *name)
{   cout << "The area of the " << name << " is " << X.area() << endl;
}
```

```
int main()
{   Circle X(6);
    show(X, "circle");
    InSquare Y(6);
    show(Y, "inscribed square");
    ExSquare Z(6);
    show(Z, "circumscribed square");
}
```

```
The area of the circle is 113.097
The area of the inscribed square is 72
The area of the circumscribed square is 144
```

A.5 模板

模板是C++提供的一种新机制，用于增强类和函数的重用性。在有模板的算法中，通常将函数声明与函数实现放在同一个文件中。

A.5.1 函数模板

1. 如何使用函数模板

对于前面讨论的函数Max，使用函数重载需要定义两个Max()函数。但是，这两个函数的函数体几乎相同，定义多个函数增加了工作量和复杂性，且容易造成不一致。通过使用模板，可以定义一个通用的Max()函数。

下述Max模板定义了一个Max()函数的家族系列，它们分别对应于不同的类型T，编译器会根据需要创建适当的Max()函数。

```
#include<iostream>
using namespace std;
template<class T>
auto Max(T x, T y)
{   return x > y ? x : y;
}
int main()
{   int i = Max(1, 2); //  创建Max<int>()函数
    double d = Max(1.1, 2.2); //  创建Max<double>()函数
    cout << i << ", " << d << endl; // 2, 2.2
    // Max(11, 2.2)歧义
}
```

对于本例，编译器会根据main()函数中Max()的调用情况创建两个Max()函数，一个的参数类型为int，另一个的参数类型为double。

2. 多参数函数模板

上述Max模板有时使用起来也不够方便，例如，对于调用“Max(11, 2.2)”，编译器不能正确推导出模板参数T代表的实际类型。因为在该调用中，一个参数的类型为int，另一个参数的类型为double。这个问题可以通过使用2个模板参数解决。

```
#include<iostream>
using namespace std;
template<class T1, class T2>
auto Max(T1 x, T2 y)
{   return x > y ? x : y;
}
int main()
{   int i = Max(1, 2); // 创建Max<int, int>()函数
    double d = Max(1.1, 2.2); // 创建Max<double, double>()函数
    cout << i << ", " << d << endl; // 2, 2.2
    auto v = Max(11, 2.2); // 创建Max<int, double>()函数, 返回double值
    cout << v << ", sizeof = " << sizeof(v) << endl; // 11, sizeof = 8
}
```

【注】这里定义的函数模板Max()在C++ 17中可以写成“auto Max(auto x, auto y) { return x > y ? x : y; }”。

A.5.2 类模板

1. 通用类

除了定义通用函数外，模板还可用于定义通用类。例如，前面介绍的Position类用于描述整数坐标。如果要求使用另一元素类型的坐标，就必须另写一个Position类。此时，可以使用模板定义一个通用的坐标类Position。

```
#include<iostream>
using namespace std;
template<class T>
class Position // 通用的坐标类Position
{   T x, y;
public:
    Position(T x = 0, T y = 0): x(x), y(y) {}
    auto GetX() const { return x; }
    auto GetY() const { return y; }
    Position &operator=(const Position &p)
    {   x = p.x, y = p.y;
        return *this;
    }
    friend bool operator==(const Position &p, const Position &q)
    {   return p.x == q.x and p.y == q.y;
    }
};
```

【注】这里省略了拷贝构造函数和析构函数的显式定义。实际上，这里重载的赋值运算符与默认的赋值运算符完成同样的工作，该重载的显式定义也可以省略。

```
int main()
{   Position<int> X(1, 2); // 或Position<int> X{1, 2}, 创建Position<int>类
    auto Y = new Position<double>{3.5, 4}; // 创建Position<double>类
    Position<int> Z;
    cout << X.GetX() << ", " << X.GetY() << endl; // 1, 2
    cout << Y->GetX() << ", " << Y->GetY() << endl; // 3.5, 4
    cout << Z.GetX() << ", " << Z.GetY() << endl; // 0, 0
    cout << (Z == X) << endl; // 0
    Z = X;
    cout << (Z == X) << endl; // 1
    delete Y;
}
```

根据main函数中的使用情况，编译器创建Position<int>类和Position<double>类。

【注】C++ 17允许根据构造函数的实际参数进行类模板参数推导。例如，上述例子中的Position<int> X{1, 2}可以写成Position X{1, 2}。这里，构造函数的两个实际参数都是整数，从而推导出类模板参数T代表的实际类型就是int。但是，Position<double>{3.5, 4}不能写成Position{3.5, 4}。因为构造函数的两个实际参数类型不同，不能推导出类模板参数T代表的实际类型。

2. 类型别名与模板别名

C++允许使用using定义类型别名和模板别名（即带有模板参数的类型别名）。

（1）基本语法。定义类型别名和模板别名的基本语法如下所示。

```
// 使用using定义类型别名，与typedef功能相同
using 类型别名 = 具体类型;
```

```
// 使用using定义模板别名，不能使用typedef
template<模板参数>
using 模板别名 = 类型模板;
```

（2）举例说明。在C++ STL中，可用priority_queue表示堆。但是，C++ STL中的priority_queue使用起来不是很方便。为了方便后续示例使用priority_queue，可以使用下述别名模板对priority_queue进行特殊化，专门定义最小堆和最大堆，并将该定义保存在文件queue.h中。

```
#pragma once // 避免重复编译
#include<queue> // priority_queue, queue
#include<vector> // vector
// 最小堆, 将STL的priority_queue特殊化, 需要大于运算
template<class T> // 使用using定义模板别名
using minheap = std::priority_queue<T, std::vector<T>, std::greater<T>>;
// 最大堆, 将STL的priority_queue特殊化, 需要小于运算
template<class T> // 使用using定义模板别名
using maxheap = std::priority_queue<T, std::vector<T>, std::less<T>>;
```

```
template<模板参数>
using 模板别名 = 类型模板;
```

A.6　C++ STL使用举例

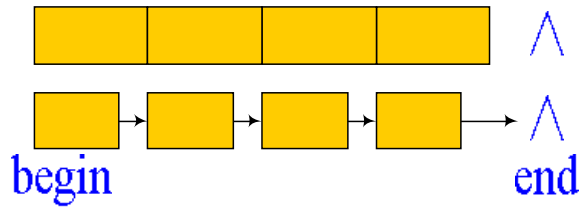
A.6.1　迭代器

1. 迭代器的含义

迭代器是指针的泛化，允许C++程序以统一的方式处理不同的数据结构（例如，数组、容器和范围等）。

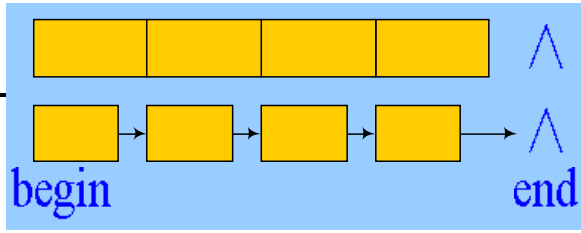
因为迭代器是指针的抽象，所以迭代器的语义是C++中大多数指针语义的概括。这确保了每个使用迭代器的函数模板都能与常规指针一起工作。

和指针一样，迭代器也代表数据元素在相应数据结构中的位置，只是不一定正好是元素地址。如图A-4所示。



图A-4　迭代器示意

通常通过一个范围遍历相应数据结构中的所有元素。其中，begin()代表首元素位置，end()表示结束位置，即最后一个元素的后一位置。



2. 迭代器的常用操作

如表A-4所示。

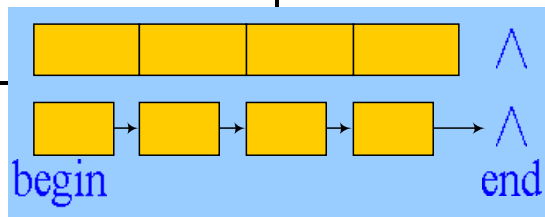
表A-4　迭代器的常用操作

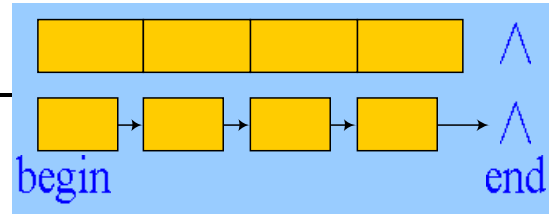
	一般(前向移动)	双向移动	连续存储
访问对象	*i*		
比较	*i*==*j*, *i*!=*j*		
修改(运算)	++*i*	--*i*	
修改(函数)	advance(*i*,*n*)	advance(*i*,-*n*)	*i*+=*n*, *i*-=*n*
	next(*i*,*n*)	prev(*i*,*n*)	*i*+*n*, *i*-*n*
距离	distance(first, last);		last-first

3. 迭代器的常用操作

下述示例通过一个数据元素连续存储的迭代器演示了迭代器的常用操作。

```
#include<vector>
#include<iostream>
using namespace std;
int main()
{   vector<int> X(10); // 10元素整数数组(连续存储)
    auto F = X.begin(), L = X.end(); // 范围
    for(auto i = F; i != L; ++i) // 正向赋值
        *i = rand() % 100;
    for(auto i = L; i != F; --i) // 逆向显示
        cout << *prev(i) << " "; // 连续存储, prev(i)可换成(i-1)
    cout << endl;
    for(auto i = F; i != L; advance(i, 2)) // 间隔输出
        cout << *i << " "; // 连续存储, advance(i, 2)可换成i+=2
    cout << endl;
    cout << *next(F, 5) << endl; // 显示X[5], next(F, 5)可换成(F+5)
    cout << distance(F, L) << endl; // distance(F, L)可换成L-F
}
```





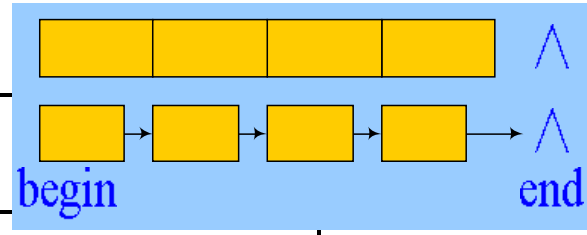
A.6.2 常用算法

1. 输出数组元素

为方便后续示例使用，这里将输出数组中前n个元素的操作组织成一个函数模板，保存在文件Print.h中。

```
//Print.h
#pragma once
#include<iostream>
```

```
template<class T> // 输出数组的前n个元素
void Print(const T X[], int n)
{   // 使用C++ STL风格实现
    auto F = X, L = F + n;
    if(L == F) return; // 没有元素
    std::cout << *F; // 首元素
    for(++F; F != L; ++F)
        std::cout << ' ' << *F; // 后续每个元素前面加上空白
    std::cout << std::endl;
}
```

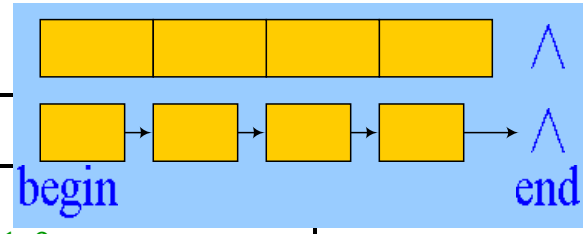


2. 常用算法使用演示

```
#include<algorithm>
#include<numeric> // accumulate, etc.
#include"print.h"
using namespace std;
```

```
int main()
{   int n = 10, X[n], Y[n];
    fill(X, X + n, 1); //  或fill_n(X, n, 1)
    //  其中, X + n和X - n的一般形式分别是next(X, n)和prev(X, n)
    cout << "fill: ", Print(X, n); // fill: 1 1 1 1 1 1 1 1 1 1
    copy(X, X + n, Y); //  或copy_n(X, n, Y);
    cout << "copy: ", Print(Y, n); // copy: 1 1 1 1 1 1 1 1 1 1
```

```
    iota(X, X + n, 0);
    cout << "iota: ", Print(X, n); // iota: 0 1 2 3 4 5 6 7 8 9
    replace(X, X + n, 0, 9);
    cout << "replace: ", Print(X, n); // replace: 9 1 2 3 4 5 6 7 8 9
    cout << "accumulate: " << accumulate(X, X + n, 0) << endl;
    // accumulate: 54
```




```
    reverse(X, X + n);
    cout << "reverse: ", Print(X, n); // reverse: 9 8 7 6 5 4 3 2 1 9
    random_shuffle(X, X + n);
    cout << "shuffle: ", Print(X, n); // shuffle: 1 8 9 7 9 4 2 6 5 3
    auto loc = find(X, X + n, 5);
    int idx = loc - X; // loc - X的一般形式为distance(X, loc)
    cout << "find 5(index): " << idx << endl;
    // find 5(index): 8
    loc = max_element(X, X + n); // min_element
    idx = loc - X;
    cout << "max_element(index, value): " << idx <<", " << *loc << endl;
    // max_element(index, value): 2, 9
    sort(X, X + n); // 默认为升序, 降序为sort(X, X + n, greater<int>())
    cout << "sort: ", Print(X, n); // sort: 1 2 3 4 5 6 7 8 9 9
    auto low = lower_bound(X, X + n, 5); // ≥5的第一个位置
    auto up = upper_bound(X, X + n, 5); // >5的第一个位置
    cout << "lower, upper(value): " << *low << ", " << *up << endl;
    // lower, upper(value): 5, 6
}
```

A.6.3 λ表达式

可以使用λ表达式达到在函数内部定义函数对象的目的。

【基本语法】C++中定义λ表达式的基本语法为

[captures](params) -> type { body }

【选项说明】

- { body }是函数体，不能省略。
- -> type是后置的返回值类型，通常省略，由return语句推断。
- (params)是参数列表，参数列表为空时，可以省略该部分。
- [captures]是外部变量列表，列出函数体需要引用的所有外部变量。该部分不能省略，函数体不需要引用外部变量时可以使用空列表[]。
- 外部变量列表通常使用形如“[=]”、“[&]”和“[*n*, &*v*]”的形式给出，其中，[]中的*n*表示使用外部变量*n*的值，&*v*表示使用外部变量*v*的引用，[=]和[&]表示允许函数体引用全部外部变量（分别使用值方式和引用方式）。

下面通过两个示例说明λ表达式的使用，其中一个示例用于演示无递归λ表达式的使用，另一个示例用于演示有递归λ表达式的使用。

1. 无递归的λ表达式

下列示例用于演示无递归λ表达式的使用，同时还演示了常用方法for_each()的使用（C++ 17可用for_each_n）。

```
#include<algorithm> // for_each, etc.
#include<iostream>
using namespace std;
int main()
{   int n = 10, X[n] = {41, 67, 34, 10, 69, 24, 78, 58, 62, 64};
    // 命名的λ表达式(显示所有奇数)
    auto odd = [](int e) // 在[]中填写需要引用的外部变量, 不能省略
    {   if(e % 2 == 1) cout << e << ' '; // 只显示奇数
    }; // 别忘了";"
    cout << "odd: ", for_each(X, X + n, odd), cout << endl; // odd: 41 67 69
```

```
    // 匿名的λ表达式(求平均值)
    double avg = 0; // 平均值
    for_each(X, X + n, [n, &avg](int e) { avg += (double)e / n; });
    // 这里, []中的n表示使用n的值, &avg表示使用avg的引用
    // [=]和[&]表示引用全部外部变量(分别使用值方式和引用方式)
    cout << "avg: " << avg << endl; // avg: 50.7
}
```

2. 有递归的λ表达式

在C++ STL中，使用类模板std::function表示函数对象。使用std::function模板类可以创建有递归的λ表达式，从而达到在函数内部定义递归函数的目的。其中，function的模板参数通常是一个省略了函数名和参数名的函数原型，该原型中的参数类型最好与λ表达式中的参数类型相同，否则调用结果难以预料。

```
#include<functional> // function, etc.
#include<iostream>
using namespace std;
int main()
{   function<void(int, char, char, char)> // 不能使用auto
    hanoi = [&hanoi](int n, char a, char b, char c) // 必须是引用(或[&])
    {   if(n <= 0) return;
        hanoi(n - 1, a, c, b);
        cout << a << "-->" << b << "\t";
        hanoi(n - 1, b, a, c);
    };
    hanoi(3, 'A', 'B', 'C');
}
```

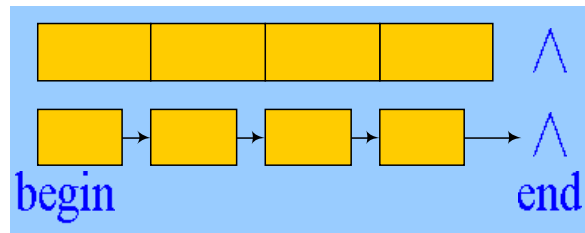
```
A-->C   A-->B   C-->B   A-->C   B-->A   B-->C   A-->C
```

A.6.4 vector

C++ STL使用类模板std::vector表示可变长的数组。下面举例说明vector的使用方法。

```
#include<algorithm> // find, etc.
#include<iostream>
#include<vector>
using namespace std;
```

```
template<class T> // 输出vector中的所有元素，形如{ 1 2 3 }
ostream &operator<<(ostream &out, const vector<T> &X)
{   auto F =begin(X), L = end(X); // 或 F = X.begin(), L = X.end()
    // begin()和end()返回迭代器，指示元素位置，类似于指针
    out << "{ ";
    for(; F != L; ++F) out << *F << " ";
    return out << "}";
}
```



```
int main()
{   int n = 3;
    vector<double> X(n), Y(n, 5), Z = {1, 2, 3, 4};
    cout << "Construct: " << X << " " << Y << " " << Z << endl;
    // Construct: { 0 0 0 } { 5 5 5 } { 1 2 3 4 }
    X = Z, Y = {1, 2, 3}, Z = {};
    cout << "=: " << X << " " << Y << " " << Z << endl;
    // =: { 1 2 3 4 } { 1 2 3 } { }
    X.assign({1, 2, 3, 4, 5}), Y.assign(n, 5);
    cout << "assign: " << X << " " << Y << endl;
    // assign: { 1 2 3 4 5 } { 5 5 5 }
    cout << "size: " << X.size() << " " << Y.size() << endl; // size: 5 3
    X.resize(4), Y.resize(4, 1);
    cout << "resize: " << X << " " << Y << endl;
    // resize: { 1 2 3 4 } { 5 5 5 1 }
    cout << "X[2]: " << X[2] << endl; // X[2]: 3 (可修改)
    cout << "front, back: " << X.front() << ", " << X.back() << endl;
    // front, back: 1, 4
```

```
    X.push_back(3), X.push_back(2);
    cout << "push_back: " << X << endl; // push_back: { 1 2 3 4 3 2 }
    X.pop_back();
    cout << "pop_back: " << X << endl; // pop_back: { 1 2 3 4 3 }
    X.insert(find(begin(X), end(X), 4), 5); //  在首个4前插入5
    cout << "insert: " << X << endl; // insert: { 1 2 3 5 4 3 }
    X.erase(find(begin(X), end(X), 3)); //  删除首个3
    cout << "erase: " << X << endl; // erase: { 1 2 5 4 3 }
    X.clear();
    cout << "clear, empty: " << X.empty(); // clear, empty: 1
}
```

A.6.5 stack

栈的插入和删除都在称为栈顶（top）的一端进行。栈的运算意味着，如果依次将元素1, 2, 3, 4插入栈，那么从栈中移出的第一个元素必定是4，即最后进入栈的元素将首先移出，因此，栈又称为后进先出（LIFO）表。

C++ STL使用类模板std::stack表示栈。下面以将数组元素反向为例说明stack的使用方法。

```
#include<stack>
#include"print.h"
using namespace std;
template<class T>
void Reverse(T X[], int n)
{   stack<T> S; // 栈
    for(int i = 0; i < n; ++i) S.push(X[i]); // 进栈
    for(int i = 0; not S.empty(); ++i) // 是否空栈
        X[i] = S.top(), S.pop(); // 栈顶，出栈
}
```

```
int main()
{    int n = 9;
     double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
     Reverse(X, n);
     Print(X, n); // 6 9 2 3 4 8 9 7 6
}
```

A.6.6 queue

队列的插入只能在称为尾部的一端进行，删除则只能在称为首部的另一端进行。队列的运算要求第一个插入队中的元素也第一个移出，因此，队列是一个先进先出（FIFO）表。

C++ STL使用类模板std::queue表示队列。下面举例说明queue的使用方法。

```
#include<iostream>
#include<queue>
using namespace std;
int main()
{   double X[] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    queue<double> Q;
    for(auto e : X)
        cout << e << " ", Q.push(e); // 进队
    cout << endl; // 6 7 9 8 4 3 2 9 6
    while(not Q.empty()) // 是否空队
        cout << Q.front() << " ", Q.pop(); // 队首，出队
    cout << endl; // 6 7 9 8 4 3 2 9 6
}
```

A.6.7　heap

1. 最小堆和最大堆

堆分为最小堆和最大堆2种，是一种带有优先级的队列。在最小堆中，队首（堆顶）元素一定是最小的元素；在最大堆中，队首（堆顶）元素一定是最大的元素。

在C++ STL中，可用priority_queue表示堆。但是，priority_queue使用起来不是很方便。为了方便使用，“A.5.2　类模板”一节的“类型别名与模板别名”部分对priority_queue进行了特殊化，专门定义了最小堆和最大堆，并将该定义保存在文件queue.hpp中。

```
// 最小堆，将STL的priority_queue特殊化，需要大于运算
template<class T> // 使用using定义模板别名
using minheap = priority_queue<T, vector<T>, greater<T>>;
```

```
// 最大堆，将STL的priority_queue特殊化，需要小于运算
template<class T> // 使用using定义模板别名
using maxheap = priority_queue<T, vector<T>, less<T>>;
```

2. 举例

这里以堆排序为例说明priority_queue的使用方法。

【注】除问题背景隐含或指定了排列顺序以外，在本书中，排序等问题涉及的排列顺序均默认为升序。

```
#include"queue.h"
#include"print.h"
```

```
template<class T> // 对数组 X[0 .. n)排序
void HeapSort(T X[], int n)
{   minheap<T> H; // 最小堆
    for(int i = 0; i < n; ++i) H.push(X[i]); // 进堆
    for(int i = 0; not H.empty(); ++i) // 是否空堆
        X[i] = H.top(), H.pop(); // 堆顶，出堆
} // 耗时O(n*log(n))
```

```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    HeapSort(X, n);
    Print(X, n); // 2 3 4 6 6 7 8 9 9
}
```

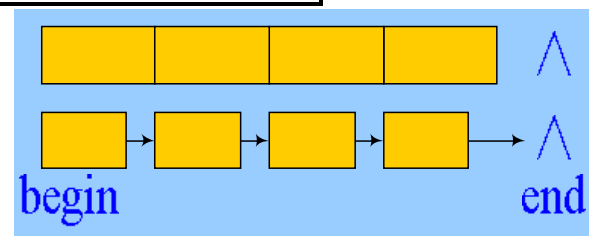
A.6.8 set

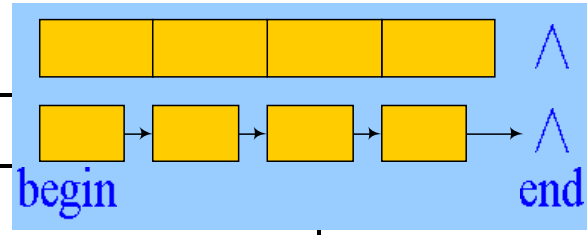
在C++ STL中，使用类模板std::set表示集合。set对象中的元素是有序的，不允许重复的。其中，第0号元素是最小元素，第1号元素是次小元素，依此类推。下面举例说明set的使用方法。

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

```
template<class T> // 输出set中的所有元素，形如{ 1 2 3 }
ostream &operator<<(ostream &out, const set<T> &X)
{   out << "{ ";
    for(const T &e : X) out << e << " ";
    return out << "}";
}
```

```
template<class T, class T2> // 在X中查找v
bool Contains(const set<T> &X, const T2 &v)
{   return X.find(v) != X.end(); // 或X.count(v) > 0;
}
```






```
int main()
{   int n = 10, w;
    set<int> X = {1, 2, 3}, Y;
    cout << "Construct: " << X << " " << Y << endl;
    // Construct: { 1 2 3 } { }
    for(int i = 0; i < n; ++i) X.insert(rand() % 15);
    Y.insert({3, 7, 17}), Y.insert(end(Y), 18); // 先尝试在end附近插入
    cout << "insert: " << X << " " << Y << endl;
    // insert: { 1 2 3 4 7 10 11 14 } { 3 7 17 18 }
    X.merge(Y); // 将Y中不属于X的元素移到X中，需C++ 17支持
    cout << "merge: " << X << " " << Y << endl;
    // merge: { 1 2 3 4 7 10 11 14 17 18 } { 3 7 }
    cout << "size: " << X.size() << " " << Y.size() << endl; // size: 10 2
    cout << "Contains(3, 13): " << Contains(X, 3) << ", "
         << Contains(X, 13) << endl;
    // Contains(3, 13): 1, 0
    cout << "begin, max value: " << *begin(X) << ", "
         << *rbegin(X) << endl;
    // begin, max value: 1, 18
```

```
    X.erase(3);
    cout << "erase(3): " << X << endl;
    // erase(3): { 1 2 4 7 10 11 14 17 18 }
    auto low = X.lower_bound(10); // >=10的第一个位置
    auto up = X.upper_bound(15); // >15的第一个位置
    cout << "lower, upper(value): " << *low << ", " << *up << endl;
    // lower, upper(value): 10, 17
    X.erase(low, up);
    cout << "erase(10 -- 15): " << X << endl;
    // erase(10 -- 15): { 1 2 4 7 17 18 }
    X.clear();
    cout << "clear, empty: " << X.empty() << " " << Y.empty() << endl;
    // clear, empty: 1 0
}
```

A.6.9 map

可以认为map是一种特殊的集合。模板类std::map<Key, T>描述可变长的std::pair<const Key, T>类型元素序列。每个对的first分量是排序键，second分量是键的关联值。可以使用first分量作为下标访问序列中的元素，返回值为相应second分量的引用，当first分量无效时会插入一个元素。下面举例说明map的使用方法。

```
#include<iostream>
#include<map>
using namespace std;
// 输出map中的所有元素，形如{ (1, 3) (2, 5) (3, 7) }
template<class K, class T>
ostream &operator<<(ostream &out, const map<K, T> &X)
{   out << "{ ";
    for(const auto &[key, val] : X)
        out << "(" << key << ", " << val << ") ";
    return out << "}";
}
```

```
int main()
{   map<double, int> X = {{1.5, 3}, {2.5, 5}, {3.5, 7}}, Y = X;
    cout << "Construct: " << X << endl;
    // Construct: { (1.5, 3) (2.5, 5) (3.5, 7) }
    cout << "X[8] = " << X[8] << endl; // X[8] = 0 (会新建X[8])
    X.insert({{7, 17}, {3.5, 8}}); // 插入，去除重复
    cout << "insert: " << X << endl;
    // insert: { (1.5, 3) (2.5, 5) (3.5, 7) (7, 17) (8, 0) }
    cout << "find(3.5): " << (X.find(3.5) != X.end()) << endl; // find(3.5): 1
    X.erase(3.5); // 删除
    cout << "erase(3.5): " << X << endl;
    // erase(3.5): { (1.5, 3) (2.5, 5) (7, 17) (8, 0) }
    int n = 3; // map数组大小
    map<double, int> W[n] = {X, Y}; // map数组
    for(int i = 0; i < n; ++i)
        cout << i << ": " << W[i] << endl;
    // 0: { (1.5, 3) (2.5, 5) (7, 17) (8, 0) }
    // 1: { (1.5, 3) (2.5, 5) (3.5, 7) }
    // 2: { }
}
```

A.7# 练习题

1. 参考C++ STL的string类构造一个自定义的字符串类。

2. 请使用C++构造一个矩阵类模板，要求至少能够完成行列数读取和元素访问等必要的基本操作。