C++程序设计

郑文立

2024年 秋季学期

第7章
间接访问—指针

- 指针的概念
- 字符串再讨论
- 指针与函数
- 指针运算与数组
- 引用类型与函数
- 动态内存分配
- 指针数组与多级指针

关于指针的几个问题

“*”的两种不同用法，勿混淆。

➢ 定义一个变量为指针

- 类型标识符 *指针变量;
- 如：int *p; 或 double *p;

➢ 使用指针所指向的对象

- 用解引用运算符“*”。如 *p 表示 p 指向的内存单元的内容。
- 在使用 *p 之前，必须先对 p 赋值。

➢ 给指针变量赋值

- int x = 5, *p = &x, *q = p;
- const char *s = “Hello”; //把字符串的起始地址存入指针s

关于加不加星号的常见情况

➢ 判断星号*的含义

- 有类型的情况下表示定义了一个指针，如 void print(char *s);
- 无类型的情况下表示取指针所指向的对象，如 y = *p;

➢ 判断要不要加星号*

- 定义一个函数的参数或返回类型为指针，要加*。
- 调用函数时将指针作为参数，不加*。
- 调用函数时将指针所指向的对象作为参数，要加*。
- 函数的返回语句return指针时不加*，return指向的对象时要加*。

地址运算符

➢ 如何获取系统分配给变量的具体地址？

- 用地址运算符“&”解决。如表达式“&x”获取的是变量x的地址。
- 赋值：intp = &x;
- **地址运算符后面不能跟常量或表达式。**
  如 &2 是没有意义的，&(m * n + p)也是没有意义的。
- **对于int *p;，用 &p 表示指针变量 p 自身的地址，是二级指针。**
- **对于int a[5];，用 &a 表示一个数组指针的值。**

指针常量与常量指针

- 指针常量：不许改变指针本身（己值和他址），本身是常量
  - 例如 int x = 5; int* const p = &x; *p = 8; //x的值变成8

- 常量指针：不许改变指针所指的对象（他值），本身是变量
  - 例如 const int *p 或 int const *p;
    int x, y; p = &x; p = &y; //p先指向x，后指向y

指针的加减

- **两个指针可以相减，但不能相加**（语法错误，编译报错）。
- 用指针操作数组元素时，注意下标越界问题。
- 怎样理解“指针+整数”的含义？以 **int** *p, k;为例，
  - p+k 表示 p[k] 的地址，即 &p[k]
  - 在数值上，如果p的值为4800，那么p+k的值为4800+**4***k
  - 两个指针相减也要在已值相减的基础上除以他型的大小

指针与数组

- 二维数组中的“指针+整数”，以int a[3][4]; 为例：
  - a是二维数组名，也是数组指针常量，指向长度为4的一维数组
  - a+k 表示 a[k] 的地址，即 &a[k]，也就是第k行的地址
  - 因此，*(a+k) 即 a[k]，表示第k行，也是第k行的一维数组名
  - a[0] 也就是第0行的一维数组名，a[0]+k 是 a[0][k] 的地址
  - 综上，*(*(a+k)+1)表示 a[k][1]
- 中括号[]的优先级高于星号*。

数组名作为函数参数

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

void f(int arr[ ], int k) {
    cout << sizeof(arr) << "    " << sizeof(k) << endl;
}

int main() {
    int a[10]={1,2,3,4,5,6,7,8,9,0};
    cout << sizeof(a) << endl;
    f(a,10);
    return 0;
}
```

输出：

40

4　　4

C++将数组名作为参数传递的情况处理成指针的传递。

动态内存分配与回收

➢ C++中由 new 和 delete 两个运算符来实现内存的动态分配。

- 运算符new用于进行内存分配：

```
p = new char;
p = new char[n];//分配数组
p = new char('A');
```

```
p = new char[n]();
p = new char[n]{'A', 0};
```

- 注意只有动态数组可以用变量来规定大小，普通数组不可以！
- 运算符delete释放new分配的内存：

```
delete p;
delete [] p;//数组的情况
```

内存分配的进一步介绍

- 静态分配：对全局变量和静态变量，编译器为它们分配空间，这些空间在整个程序运行期间都存在。
- 自动分配：非静态的局部变量的空间分配在系统栈区(stack)。当进入程序块（通常是函数被调用）时，空间被分配；当程序块执行结束后，空间被自动释放。
- 动态分配：在程序执行过程中需要新的存储空间时，可用动态分配的方法向系统申请（即new），当不再使用时显式地还给系统（即delete，非自动）。这部分空间是从被称为堆(heap)的内存区域分配的。

内存泄漏

- 内存泄漏通常是由于作为局部变量的指针在函数返回后（或程序块结束后）消亡了，而在此之前没有用delete释放动态内存。
- 由于你失去了指向动态分配的区域的指针，因此无法再访问这些区域。但堆管理器认为你在继续使用它们，因此不能分配给其它变量。
- 当用delete释放了内存区域后，**指针的值不受delete影响，仍然指向堆区域**，但你不能再对指针解引用来访问已被释放的区域。
- 要确保在程序中同一块用new分配的内存只释放一次。

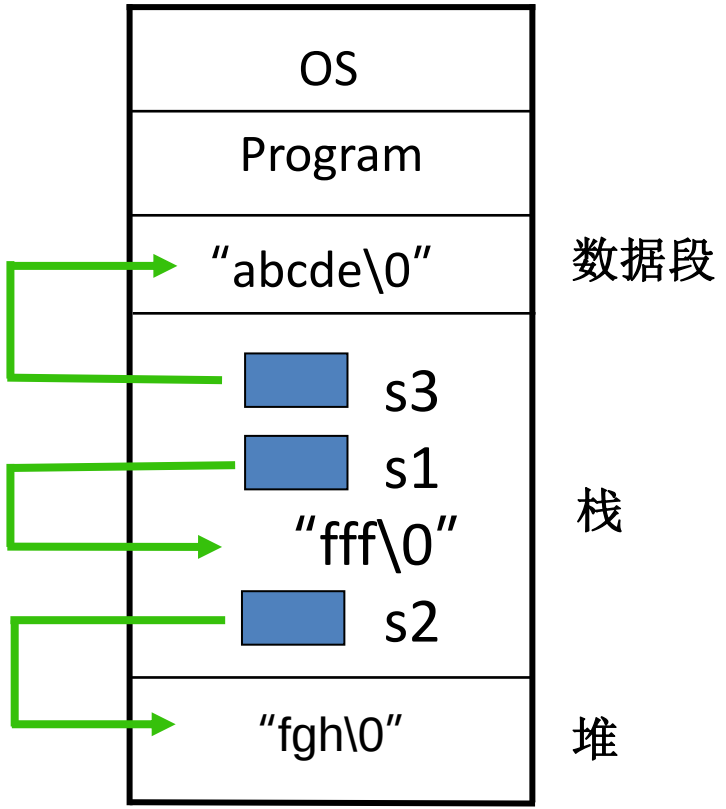
字符串再讨论

下列语句的执行结果：

```
int main() {
  char  *s1, *s2;
  const char *s3 = “abcde” ;
  char  ch[] = “fff” ;
  s1 = ch; s2 = new char[10];
  strcpy(s2, “fgh” );
  return 0;
}
```

给指针赋初值

给指针所指的对象赋值



引用参数

- C++引入引用的主要目的是将引用作为函数的参数。

指针参数

```
void swap(int *m, int *n) {
    int temp;
    temp=*m;
    *m=*n;  *n=temp;
}
```

调用：swap(&x, &y)

引用参数

```
void swap(int &m, int &n) {
    int temp;
    temp=m;
    m=n; n=temp;
}
```

调用：swap(x, y)

m和n分别是x和y的别名

- 注意：函数使用引用参数时，实参必须是变量，不能是一个表达式。

使用引用的注意事项

- 利用引用传递参数的好处是减少函数调用时的开销。
  - 不需要分配新的内存空间
  - 不需要复制实际参数的数值
- 引用不是一种数据类型，下面的“类型说明”都是非法的。

```
int &&r= ……; //不能建立引用的引用（&&表示“且”）
int &a[3] = ……; //不能建立引用的数组
int &*p = ……; //不能建立引用的指针（但可以有指针的引用）
```

```
int *&p = q;
```

返回指针或引用

- 当函数返回指针时，不能返回该函数的局部变量的地址。
- 当函数返回引用时，不能返回该函数的局部变量的引用。
  - 包括不能返回静态局部变量的地址或引用（规范性问题）。

- 主调函数中被返回值赋值的变量通常与函数的返回类型相同。
  - 但也可以用普通变量来接收函数返回的引用，例如：

```
int x = myfun(); // myfun的返回类型为int &
```

返回引用的用途

- 将函数用于赋值运算符的左边(左值)。

```
// file1.cpp

int &index(int); //声明返回引用的函数

void main() {
    index(2) = 25; //通过接口给a[2]赋值
    cout << index(2); //通过接口输出a[2]
    return;
}
```

```
// file2.cpp

// 被封装的全局变量数组a
int a[ ] = {1, 3, 5, 7, 9};

// 数组a的接口函数
// 该函数返回a[j]的引用（别名）
int &index(int j) {
    assert( j >= 0 && j < 5);
    return a[j];
}
```

指针数组与数组指针

- 一维指针数组的定义：类型名　*数组名[指针个数];
- 例如，char *str[10]；定义了一个名为str的指针数组，该数组有10个元素，每个元素是一个指向字符的指针（即char*类型）。

- 一维数组指针的定义：类型名　(*指针变量名)[一维数组的长度];
- 注意：**数组指针**不是指针数组，圆括号不能省略，如果省略了圆括号就变成了指针数组，即定义了多个指针。

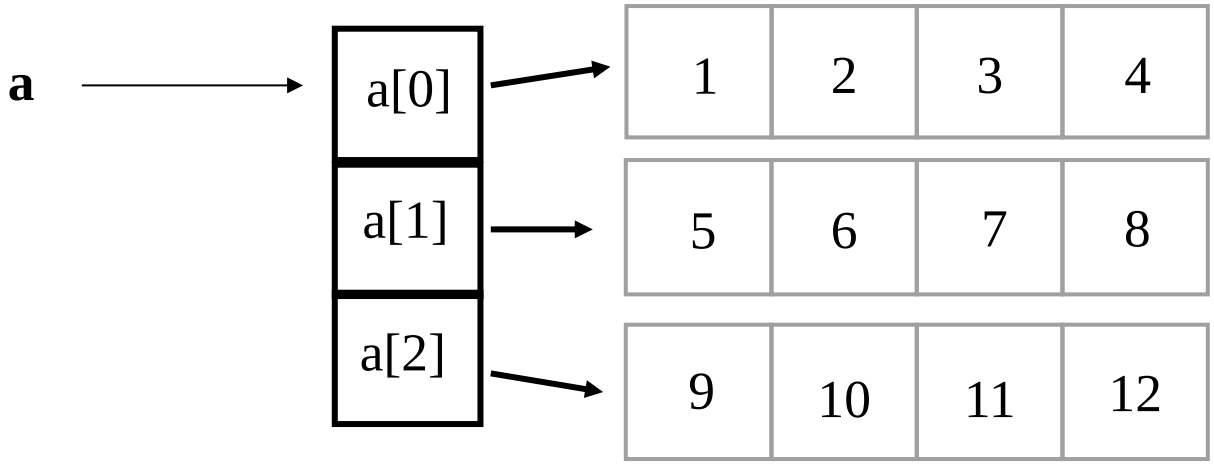
多级指针

- 如有定义char *str [10]；则str数组的每个元素都是一个类型为char*的一级指针，都可以通过二级指针来访问。
- 如有定义char **p；则可以将str的某个元素的地址赋给p，使p指向一个一级字符指针，即 p = &str[i];
- 不同级的指针不能相互赋值。

动态二维数组

- `int **a; a = new int*[3]; // 创建了3个指针变量`

  `for(int i=0; i<3; ++i) a[i] = new int[4]; //别忘了这步!`



assert宏

- 使用assert宏需要包含标准库cassert。
- assert()有一个参数，通常是一个表达式，称为“断言”。

  如果断言为真，则程序继续运行。

  如果断言为假，则在发出一个“错误消息”后终止程序。
- 该“错误消息”包含源文件名、出错行数和表达式的字面内容，十分有助于debug。

第8章 数据封装—结构体

结构体的概述

结构体类型的定义

结构体类型的变量

结构体数组

结构体作为函数的参数

结构体类型的定义

- 定义结构体类型中包括哪些分量。
- 格式：

```
struct 结构体类型名{
    字段声明;
};
```

如：

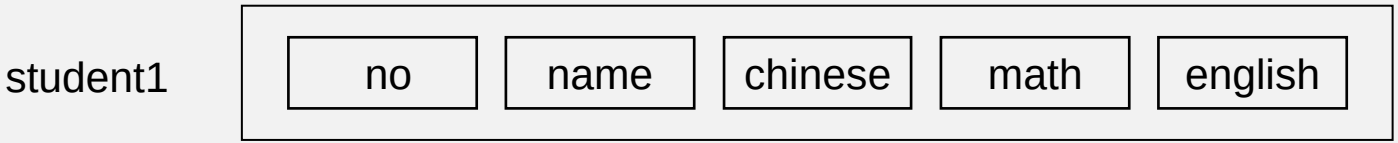
```
struct studentT {
    char  no[10];
    char  name[10];
    int chinese;
    int math;
    int english;
};
```

← 有分号

只在定义结构体类型时有struct！定义结构体变量或形参时都没有

结构体变量的定义和初始化

- 结构体变量的定义和普通的变量定义一样。如定义了结构体类型studentT，就可以定义结构体变量：studentT　student1;
- 一旦定义了一个结构体类型的变量，系统在分配内存时就会分配一块连续的空间，按照字段声明的顺序依次存放它的每一个分量。



- 结构体变量的初始化：

studentT student1 = { “00001”， “张三” ，87，90，77 };

通过指针操作记录

- **首先必须要给结构体指针赋值**，如：

  sp = &student1;

- 结构体指针的引用：

  (*指针).成员　　如：(*sp).name

  指针->成员　　如：sp->name

  } **student1.成员**

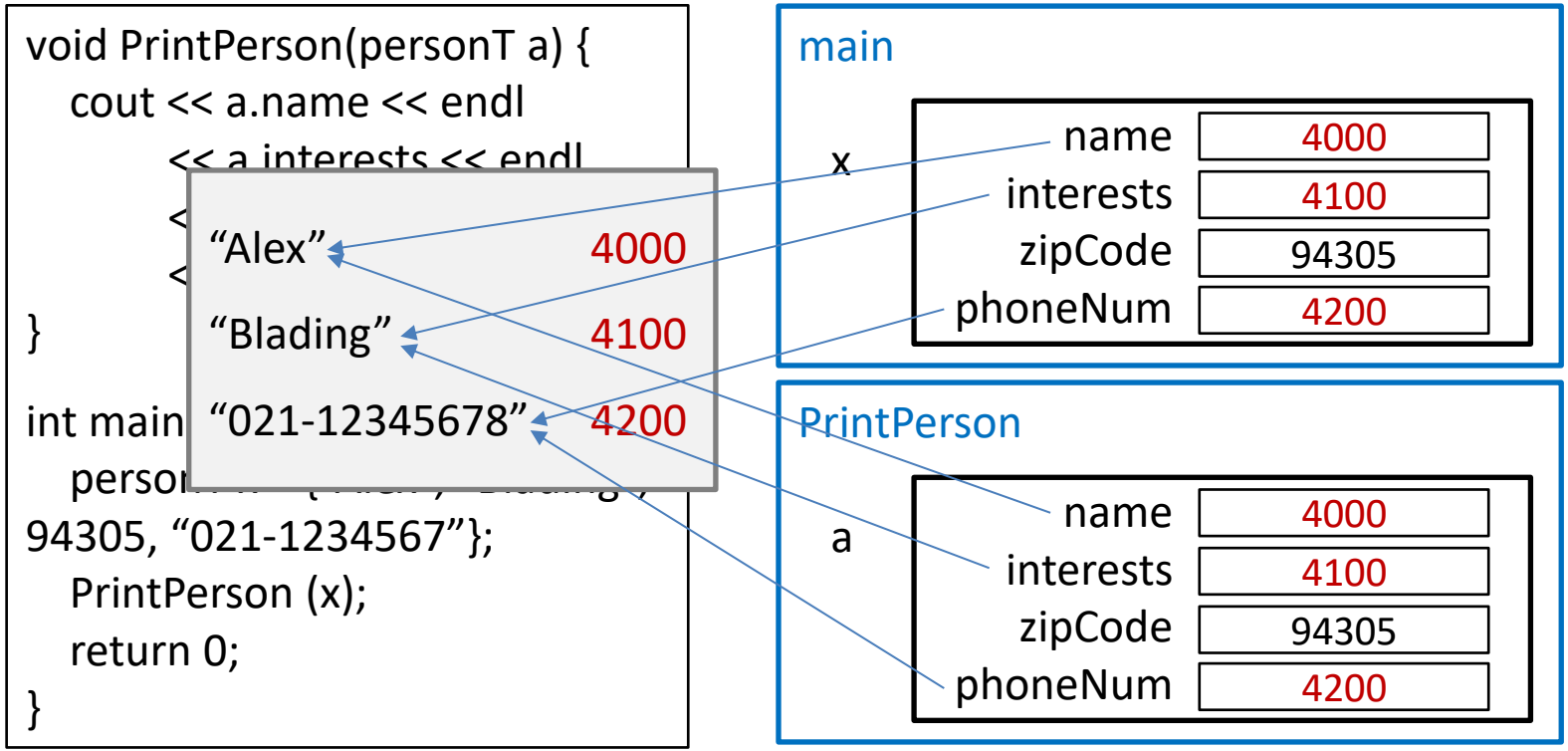
- ->是所有运算符中优先级最高的。
- 通常程序员习惯使用第二种方法。

结构体变量的赋值

- 同类型的结构体变量之间可以相互赋值，如student1 = student2;将student2的成员对应赋给student1的成员。
  - 本质上是把student2在内存中的每个字节按顺序复制到student1的内存空间里（而不是对每个成员分别进行一次赋值）。
  - 因此，即使结构体成员中包含数组，该赋值操作也没问题，数组里的每个字节会一一对应赋值（而不是对数组名赋值）。
  - 但动态数组是另一回事，因为动态数组不是成员（指针才是）。
- 不能用 == 或 != 比较两个结构体变量是否相同。
  - 如果要判断两个结构体变量是否相同，必须逐个比较其成员。

结构体作为函数的参数

结构体的值传递

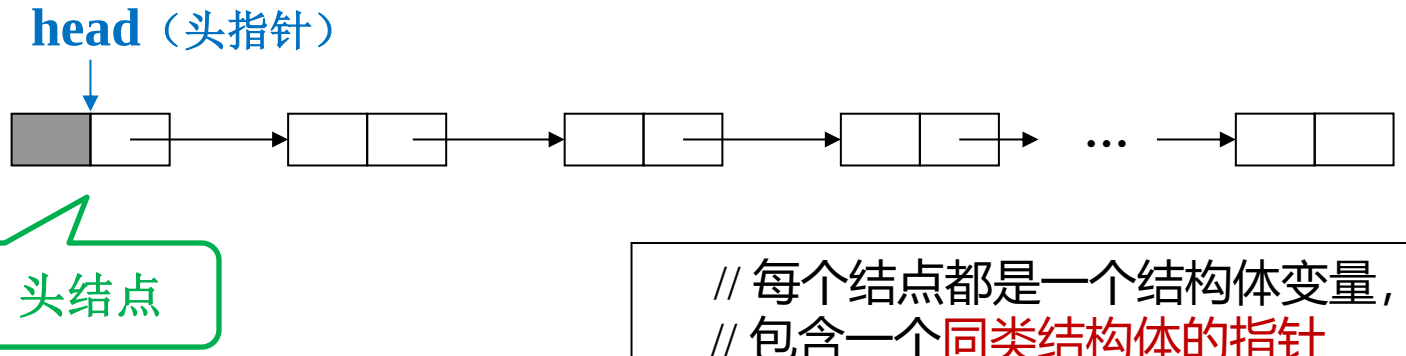


指向结构体的指针作为参数

- 如果希望把函数内部对结构体的修改返回给主调函数，可以用指针传递或引用传递。
- 即使不需要返回对结构体的修改，但由于结构体一般占用的内存量都比较大，值传递既浪费空间又浪费时间，因此实际编程中往往使用指针传递或引用传递（尤其是后者）。
- 指针或引用传递的风险是函数中可以修改实际参数，如果要控制函数中不能修改实际参数，可以在形式参数中加const限定。

链表

➢ 单链表：只指出后继关系的链表。




```
// 每个结点都是一个结构体变量，
// 包含一个同类结构体的指针
struct  linkRec
{
    datatype  data;
    linkRec  *next;
}
```

链表的操作：建立

```
head = new  linkRec; // 头指针
rear = head; // 尾指针
cin >> in_data;
while(输入未结束) {
    p = new linkRec; //建立一个新结点
    p->data = in_data; //在新结点中存入数据
    rear->next = p; //把新结点连接到链表末尾
    rear = p; //把链表末尾更新为新结点
    cin >> in_data;
}
rear->next = NULL; //设置链表结束标志
```

链表的操作：插入与删除

```
linkRec *tmp = new linkRec;   // 创建一个新结点
tmp->data = x;                // 存入数据到新结点的数据成员中
tmp->next = p->next;          // 把新结点和p结点的下一结点相连
p->next = tmp;                // 把p结点和新结点连接起来
```

```
linkRec *d = p->next;         // 找到要删除的d结点（即p的下一个）
p->next = d->next;            // 把p结点和d的下一个结点连接起来
delete d;                     // 删除d结点
```

第9章
模块化开发

自顶向下的分解

模块划分

库的设计与实现

库的应用

头文件的设计

- 为方便起见，我们把所有的**符号常量定义、类型定义和函数原型声明**写在一个头文件中，让每个模块都include这个头文件。
- 这样一来，每个模块就不必再写那些常量和类型的定义以及函数声明了。

```
// File: main.cpp
#include “header.h”

int main() {
    float a;
    cin >> a;
    cout << yourfun(a);
    cout << endl;
    cout << myfun(a);
    return 0;
}
```

```
// File: source.cpp
#include “header.h”

char myfun(float x) {
    int result = yourfun(x);
    return result%10 + ‘0’;
}

int yourfun(float x) {
    return x+0.5;
}
```

```
// File: header.h
#include <iostream>
using namespace std;

char myfun(float);
int yourfun(float);
```

编译源文件时，只匹配函数调用与函数声明；

连接时才查找函数定义。

头文件的设计

- 但使用头文件又会引起另一个问题：
  如果头文件A包含了头文件B，且另一个模块包含了这两个头文件，编译器会发现头文件B中的类型定义、符号常量和函数原型的声明在程序中重复出现，引起编译错误。

- 解决方法：
  需要用到一个新的编译预处理命令：

```
#ifndef    标识符
   …
#endif
```

```
#ifndef  _name_h
#define _name_h
   //头文件真正需要写的内容：
   //符号常量定义
   //新类型定义（结构体、枚举等）
   //函数原型声明
#endif
```

模块划分

- 把程序再分成几个小的源文件。每个源文件都包含一组相关的函数。**一个源文件被称为一个模块**。
- 模块划分标准：块内联系尽可能大，块与块之间联系尽可能小。
- 模块的**内部状态**：
  - 该模块中的所有函数都可以使用。
  - 其它模块中的所有函数都不能使用。
- 所以通常用**全局变量**来定义模块的内部状态。

设计自己的库

- 一个库应该有一个主题。一个库中的函数都应该是处理同一类问题。如标准库iostream包含输入输出功能，cmath包含数学运算函数。我们自己设计的库也要有一个主题。
- 设计一个库还要考虑到他的通用性。库中的功能应来源于某一应用，但不局限于该应用，而且要高于该应用。在某一应用程序中提取库内容时应尽量考虑到兼容更多的应用，是其他应用程序也能共享这个库。

库的设计与实现

- 库的**接口**：
  - 库的用户必须了解的内容，包括库中函数的原型、这些函数用到的符号常量和自定义类型；
  - 接口文件通常为头文件。
- 库的**实现**：
  - 定义函数和全局变量（类的实现就是定义成员函数与静态数据成员）。
  - 实现文件可以为源文件（静态链接）或二进制文件（动态链接）。
- 库将接口与实现分离的设计方法称为信息隐藏。

第10章
创建功能更强的对象

从面向过程到面向对象

常量对象与常量成员函数

类的定义

常量数据成员

对象的使用

静态数据成员与静态成员函数

对象的构造与析构

友元








面向对象的程序设计的特点

- 代码重用：圆类型可以让所有需要处理圆的程序员使用。
- 实现隐藏：
  - 类的使用者收集已有的工具快速解决所需解决的问题；
  - 这些工具是如何实现的，类的使用者不需要知道。
- 继承：在已有工具的基础上加以扩展，形成一个功能更强的工具。
- 多态性：处理继承层次结构时，可把各层次的对象看成根类对象。
  - 只要定义一个根类指针，就可以操作所有类型的对象。
  - 调用语句可以不受新增类型的影响，不用修改，但功能得到了扩展。

类定义的说明

```
class  类名 {
    private: //可写可不写
     私有数据成员和成员函数
    public:
     公有数据成员和成员函数
};
```

- private和public的出现次序可以是任意的。也可以反复出现多次。
- 成员还可以被说明为protected。
- 数据成员一般说明为private，需要被用户调用的成员函数说明为public。
- 不能让public成员函数返回private数据成员的引用（规范性问题）。
- 可以将成员函数的定义直接写在类定义中，这些函数被默认为内联函数。

对象赋值语句

- 同类型的对象之间可以互相赋值，如两个有理数对象r1和r2，可以执行 r1 = r2（和结构体类型的变量之间互相赋值是一样的）。
- 对象之间的赋值与拷贝构造无关，需注意避免混淆。

```
Rational r1 = r2; // 拷贝构造
Rational r1; r1 = r2; // 先构造，后赋值，没有拷贝构造
```

对象的使用

- 在程序运行过程中，如果通过对象x来调用了成员函数f，那么x就是f在这次执行时的“当前对象”。

```
//add函数将r1和r2相加，结果存于当前对象
void Rational::add(const Rational &r1, const Rational &r2) {
    num = r1.num * r2.den + r2.num * r1.den;
    den = r1.den * r2.den;
    ReductFraction(); //化简
}
```

```
//例：add函数的调用
Rational r1, r2, r3;
//中间省略给r1和r2赋值的操作
r3.add(r1, r2);
```

this指针

- 每个成员函数都有一个隐藏的指向本类型的指针形参this，它指向此次调用成员函数的当前对象。
- 在成员函数中，this是当前对象的地址，*this是当前对象。
- 在写成员函数时一般省略this，编译时会自动加上它们。

• 例如，对成员函数

```
void create(int n, int d) { num = n; den = d;}
```

经过编译后，实际函数为

```
void create(int n, int d) { this->num = n; this->den = d;}
```

构造函数的使用

- 有了构造函数后，对象定义的一般形式为：

  类名 对象名（实际参数表）；

- 实际参数表必须和该类的某一个构造函数的形式参数表相对应。

- 如果这个类有一个构造函数是没有参数的，那么还可以用：

  类名 对象名；

- 不带参数的构造函数都称为默认（缺省）的构造函数。
- 仅在没有定义构造函数的情况下，编译器才会自动生成默认的构造函数。
- 默认的构造函数不一定是编译器自动生成的。

动态对象的初始化

- 动态变量（这里指系统内置类型的变量）的初始化是在类型后面用一个圆括号指出它的实际参数表（一般是一个参数）。
- 类似的，如果要为一个动态的IntArray对象指定其数组下标范围为20到30，可用下列语句：

```
p = new IntArray(20, 30); // 调用构造函数，构造动态对象
```

- 括号中的实际参数要和构造函数的形式参数表相对应，否则会因无法调用构造函数而引发编译报错。

使用初始化列表的原因

- 我们完全可以在函数体内对数据成员赋值（但这不是初始化）。
- 事实上，不管构造函数中有没有构造函数初始化列表，在执行构造函数体之前，系统都会先初始化每个数据成员。
- 在构造函数初始化列表中没有提到的数据成员，系统会用该数据成员对应类型的默认构造函数对其初始化（也就是初值为随机值）。
- 显然利用初始化列表可以提高构造函数的效率。在初始化的时候，同时完成了赋初始值的工作。

必须用初始化列表的情况

- 当数据成员不是普通的内置类型，而是某一个类的对象，可能无法直接用赋值语句在构造函数体中为它赋值。如果该对象成员需要被赋特定的初始值，则必须用初始化列表调用其构造函数。例如：

```
Complex(int r1, int r2, int i1, int i2): real(r1, r2), imag(i1, i2){}
```

- 类包含了一个常量的数据成员，常量只能在定义时对它初始化，而不能对它赋值（构造函数的函数体中只能给数据成员赋值，只有初始化列表才能给数据成员初始化）。因此也必须放在初始化列表中。

拷贝构造函数的应用

- 拷贝构造函数的应用场合：①定义对象时；②把对象作为参数传给函数时；③把对象作为返回值时（②和③都是指值传递）。
- 拷贝构造函数用于定义对象时有两种用法：直接初始化和拷贝初始化。

- 直接初始化且圆括号中的实参为同类对象，
  如 IntArray arr2(arr1);
- 拷贝初始化是用“=”符号进行初始化，
  如 IntArray arr1 = IntArray(20,30);
  IntArray arr2 = arr1;

把对象作为参数传给函数时

- 如有函数：void f (IntArray array2); //注意参数是对象且进行值传递
- 函数调用：f (array1);
- 本质上是在被调函数中定义了一个局部对象array2作为形参，并用主调函数中已有的实参对象array1来初始化这个新定义的形参对象。所以会调用拷贝构造函数。

- **相当于执行了定义语句：IntArray array2 = array1;**

把对象作为返回值时

- 如有函数

```
IntArray f () {
    IntArray a;
    …
    return a;
}
```

- 当执行到return语句时，会在主调函数中创建一个IntArray类的临时对象作为返回值，并调用拷贝构造函数，用对象a来初始化该临时对象。
- **相当于执行了定义语句：IntArray 临时对象 = a;**

移动构造函数

- 如有函数IntArray f()与函数调用IntArray arr = f();
- 因为是定义对象arr并将其初始化的语句，所以要调用构造函数。
- 由于作为实参的同类对象是个临时对象，所以不调用拷贝构造函数，而调用移动构造函数，使arr直接获得临时对象的值。

析构函数

- 不要自己写一行代码来调用析构函数！
- 在对象消亡时，由编译系统自动调用析构函数。
  - 可以是用delete显式地使动态对象消亡（堆区），
  - 也可以是局部对象在程序块结束时自动消亡（栈区）。
- 析构函数的“本职工作”是释放所有数据成员的内存空间（这是系统自动执行的，不是程序员写的）。
- 当有数据成员是指针且指向堆空间的时候，为了避免内存泄漏，析构函数应该把这块堆空间也释放掉（这是要程序员写在析构函数里的）。

常量（const）对象

- 常量对象的定义：const 类名 对象名（参数表）；
- 例如：const IntArray arr1(20, 30);
- 常量对象不能被赋值，只能初始化，而且**一定要初始化**。

- 如何保证数据成员不被修改?
- 数据成员一般都由成员函数修改。当定义了一个常量对象后，为保证成员函数不修改数据成员，**C++规定：常量对象只能调用常量成员函数**。

常量成员函数

- 如果在编写常量成员函数时，不慎修改了数据成员，或者调用了非常量成员函数，编译器将指出错误，这无疑会提高程序的健壮性。

```
class A {
        int x;
    public:
        A(int i) { x = i; }
        int getx() const { return x; }
};
```

注意const的位置

```
class A {
        int x;
    public:
        A(int i) { x = i; }
        int getx() const;
};
```

静态数据成员

- 静态数据成员不属于对象的一部分，而是类的一部分；
- 静态数据成员的初始化不能放在类的构造函数中；
- 我们知道类的定义并不为数据成员分配空间，数据成员的空间是在定义对象时分配的。但静态数据成员属于类而不属于对象，因此定义对象时不为静态数据成员分配空间（所以用sizeof来获取一个类对象占用的字节数时，不计入其静态数据成员）。

静态数据成员的定义

- 为静态数据成员分配空间称为静态数据成员的定义。
- 如 double SavingAccount::rate = 0.05; //注意此处没有static
- 该定义为静态数据成员rate分配了空间，并给它赋了一个初值0.05。
- 如果没有这个定义，连接时会报告一个错误。**别忘啦！**

静态数据成员的使用

- **可以通过“类名::”来访问静态数据成员。**
  如： SavingAccount::rate //须符合访问权限的限制
- 但从每个对象的角度来看，它似乎又是对象的一部分，因此又可以从对象调用它。如有个SavingAccount类的对象obj，则可以用：obj.rate
- 由于静态数据成员是整个类共享的，因此不管用哪种调用方式，得到的值都是相同的。

静态成员函数

- 可以通过“类名::”来调用静态成员函数。
- 编译器**不会为静态成员函数加上this指针**。
  因此，静态成员函数没有“当前对象”这一概念。
- 只能访问类中的静态成员，无法处理类中的非静态成员。
  - 因为非静态成员是属于对象的，必须要有this指针来指定对象。
  - 但如果参数传入类的对象，则仍可以访问参数的非静态成员。
- 静态成员函数的声明只需要在类定义中的函数原型前加上保留词static。如果在类外定义静态成员函数，在定义时不加static。

友元函数

- 一个友元函数就是一个普通的函数（或称全局函数），如果不声明为友元，那么它和这个类毫无关系。
- 友元函数自身的定义可以写在类外部，也可以写在类内部。
  - 注意：即使把友元函数的定义写在类内部，它也不是成员函数。
- 用关键词 **friend** 说明友元函数，使它能访问类的私有成员。
- 写了友元函数的声明之后就不用另外再写该函数的声明语句了。

友元成员

- 友元成员是指其它某个类的成员函数。
- 它可以访问friend声明语句所在类的私有成员和公有成员。
- **类A的成员函数作为类B的友元函数时，必须先声明类B，再定义类A，然后定义类B。**
- 格式：

  friend　函数返回类型　类名标识符::函数名(参数列表);

友元类

- 整个类作为另一个类的友元。
- 当A类被说明为类B的友元时，类A的所有函数都是类B的友元函数。
- 声明方法：

```
class Y{ …… };
class X{
    ……
    friend class Y;
    ……
};
```

第11章
运算符重载

什么是运算符重载

运算符重载的方法

几个特殊的运算符的重载

自定义类型转换运算符

运算符重载实例

运算符重载的限制

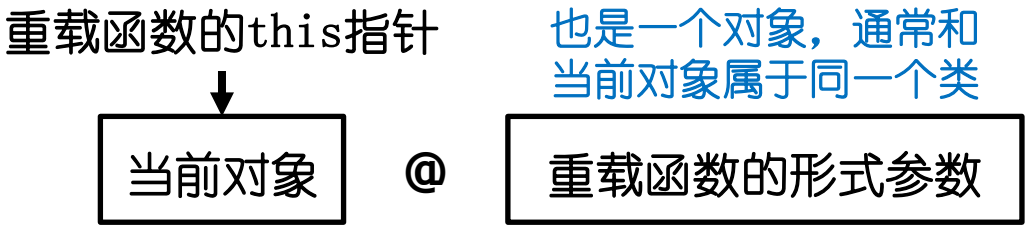
- 不是所有的运算符都能重载。
- 重载不能改变运算符的优先级和结合性。
- 重载不能改变运算符的操作数个数。
- 不能创建新的运算符。

函数原型

- 运算符重载函数的形参个数与该运算符的操作数的个数相同。
- 运算符重载可以通过类的成员函数来实现，也可以通过全局函数来实现。
  - 当重载成全局函数时，通常需要把此函数设为友元函数。
  - 当重载成类的成员函数时，它的形式参数个数比运算符的运算对象数少1。这是因为编译器会把this指针作为运算符的第一个参数。
- 赋值（=）、下标（[]）、函数调用（()）和成员访问（->）只能重载为成员函数，而输入（>>）和输出（<<）只能重载为全局函数。

函数原型

- 当把一个一元运算符重载为成员函数时，该函数没有形式参数。
- 当把一个二元运算符重载为成员函数时，该函数只有一个形式参数，就是右操作数，而当前对象是左操作数。



运算符重载

- 为Rational类增加“+”和“*”的重载函数，用以替换现有的add和multi函数

方案一：重载为成员函数：

运算符重载函数的返回类型不是与运算符本身绑定的，要根据实际需求来设计。

```
class Rational {
   private:
            int num;
            int den;
            void ReductFraction();
   public:
            Rational(int n = 0, int d = 1) { num = n; den = d;}
            Rational operator+(const Rational &r1); //+运算符重载
            Rational operator*(const Rational &r1); //*运算符重载
            void display() { cout << num << '/' << den;}
}
```

下标运算符重载

- 需求：能否像普通的数组那样通过下标来操作IntArray类的对象中的数组元素？这样可以使IntArray类更像一个功能内置的数组。
- 取代IntArray类的成员函数insert和fetch。
- 可以通过重载下标运算符（[]）来实现。

```
int & IntArray::operator[](int index) {
    if (index < low || index > high) { cout << "下标越界"; exit(-1); }
    return storage[index - low];
}
```

重载函数的原型设计考虑

➢ 参数设计：

- 对于任何运算符重载函数的参数，如果仅需要读取参数，而不改变它，一般用常量引用来传递。
- 只有对于需要修改操作数的情况，对应的参数通常用非常量引用来传递。

➢ 返回值的类型设计：

- 如果运算符的结果产生一个新值，就需要产生一个作为返回值的新对象。
- 对于逻辑运算符，一般应得到一个int或bool的返回值。
- 所有的赋值运算符（如=、+=等）均改变左值，一般应该返回一个刚刚改变了的左值的非常量引用。

赋值运算符

- 对任意类，如果用户没有自定义赋值运算符函数，那么系统为其生成一个缺省的赋值运算符函数，在对应的数据成员间赋值。
- 当类含有类型为指针的数据成员时，可能会带来一些麻烦。以IntArray类的对象赋值arr1=arr2为例，该赋值操作
  - 会引起内存泄漏，并使这两个数组的元素存放于同一块空间中。
  - 当这两个对象析构时，先析构的对象会释放存储数组元素的空间。而当后一个对象析构时，无法释放存放数组元素的空间。

IntArray类的赋值运算符重载函数

```
IntArray &IntArray::operator=(const IntArray & a) {
    if (this == &a) return *this; //防止自己复制自己
    delete [ ] storage; //归还空间
    low = a.low;
    high = a.high;
    storage = new int[high – low + 1]; //重新申请空间
    for (int i=0; i <= high – low; ++i) //复制数组元素
            storage[i] = a.storage[i];
    return *this; //赋值运算符重载函数一般返回当前对象自身的引用
}
```

“++”和“--”重载

- 这两个操作符可以是前缀，也可以是后缀。而且前缀和后缀的含义是有区别的。所以，必须有两个重载函数。
- C++规定：＋＋和--作为前缀是一元操作符，作为后缀是二元操作符。

- 成员函数：
  - ++ob重载为：ob.operator++()
  - ob++重载为：ob.operator++(int)

编译器用0作为这两个int参数的值

- 友元函数：
  - ++ob重载为：operator++(X &ob)
  - ob++重载为：operator++(X &ob, int)

输入输出运算符重载

- >>和<<必须被重载成全局函数。
- 因为左操作数是输入/输出流对象，无法调用自定义类的成员函数。

```
ostream & operator<<(ostream &os, const ClassType &obj)
{
    os << 要输出的内容;
    return os;
}
```

当用cout<<调用该重载函数时，此处的os即成为在被调函数内使用的cout的别名。

左操作数（第一个形参）不能加const

输入输出运算符重载

- >>和<<必须被重载成全局函数。
- 因为左操作数是输入/输出流对象，无法调用自定义类的成员函数。

```
istream & operator>>(istream &is, ClassType &obj)
{
   is >> 要输入的内容;
   return is;
}
```

当用cin>>调用该重载函数时，此处的is即成为在被调函数内使用的cin的别名。

两个形参都不能加const

自定义类型转换运算符

内置类型到类类型的转换

原型为 Rational (int n=0, int d=1)

- 利用单实参的构造函数来转换。
- 例如，对于Rational类的对象r，可以执行 r = 2;
- 此时，编译器隐式地调用Rational的构造函数，传给它一个参数2。构造函数将构造出一个num=2，den=1的Rational类的对象，并将它赋给r。
- 也就是说相当于 r = Rational(2); //第2个形参默认值为1

自定义类型转换运算符

类类型到内置类型或其他类类型的转换

- 可以通过类型转换函数实现。
- 类型转换函数必须重载为成员函数。
- 类型转换函数的格式

```
operator 目标类型名 ( ) const {
  ……
  return (结果为目标类型的表达式);
}
```

- 类型转换函数的特点：
  - 无参数，无返回值
  - 是常量成员函数

```
Rational::operator double () const {
   return  double(num)/den;
}
```

第12章 组合与继承

组合

继承（派生类）

- 单继承的格式
- 基类成员在派生类中的访问特性
- 派生类对象的构造、析构与赋值操作
- 重定义基类的函数
- 派生类作为基类

抽象类

多态

组合

- 组合就是把用户定义类的对象作为新类的数据成员。
  - 我们把这种数据成员称为“对象成员”
  - 组合是代码重用的一种方法
  - 组合的目的是用“小工具”快速组装“大工具”
- 组合表示一种聚集关系，是一种部分和整体的关系。
- **用初始化列表去初始化“对象成员”。**

Complex 类的定义

```
class Complex{
    Rational real;      //实部
    Rational imag;      //虚部
  public:
    Complex(int r1 = 0, int r2 = 1, int i1= 0, int i2 = 1) : real(r1, r2), imag(i1, i2) { }
    Complex(const Rational &r1, const Rational &r2) : real(r1), imag(r2) { }
    void add(const Complex &c1, const Complex &c2) {
        real.add(c1.real, c2.real);   imag.add(c1.imag, c2.imag);
     }
    void display() { real.display(); cout << '+'; imag.display(); cout << 'i'; }
};
```

调用Rational的拷贝构造函数

调用Rational的普通构造函数

派生类对基类成员的访问

- 派生类的成员函数不能访问基类的私有数据成员。
- protected访问特性：

- protected成员是一类特殊的私有成员，它不可以被全局函数或其他类的成员函数访问，但**能被派生类的成员函数访问**；
- protected成员破坏了类的封装，基类的protected成员改变时，所有派生类都要修改。

继承（派生类）

基类成员的访问说明符	继承类型 public继承	protected继承	private继承
public	在派生类中为public 可以由任何非static成员函数、友元函数和非成员函数访问	在派生类中为protected 可以直接由任何非static成员函数、友元函数访问	在派生类中为private 可以直接由任何非static成员函数、友元函数访问
protected	在派生类中为protected	在派生类中为protected	在派生类中private
	可以直接由任何非static 成员函数、友元函数访问		
private	在派生类中隐藏	在派生类中隐藏	在派生类中隐藏
	可以通过基类的public 或protected成员函数或非static成员函数和友元函数访问		

影响多态性，一般不用

派生类的构造函数和析构函数

- 由于派生类继承了其基类的成员，所以在建立派生类的对象时，必须初始化从基类继承的数据成员。派生类对象析构时，也必须析构从基类继承的数据成员。
- 派生类不继承基类的构造函数和析构函数（因为类名不同），但是派生类的构造函数和析构函数能调用基类的构造函数（显式或隐式）和析构函数（隐式）。

构造函数的格式

- 派生类构造函数的格式：

  派生类构造函数名（参数表）：基类构造函数名（参数表） {……}

- 如果派生类新增的数据成员中含有对象成员，则在创建对象时，先执行基类的构造函数，再执行成员对象的构造函数，最后执行自己的构造函数体。
- 如果在初始化列表中有多个派生类新增的数据成员（包括对象成员），那么它们的构造顺序与在初始化列表中的顺序无关，而取决于它们在类定义中声明的顺序。

派生类对象的析构

- 派生类的析构函数只处理自己新增的数据成员，从基类继承的数据成员由基类的析构函数来处理。
- 派生类析构函数会自动调用基类的析构函数。
- 派生类对象析构时，先执行派生类的析构函数，再执行基类的析构函数（析构的顺序与构造的顺序正好相反）。

重定义基类的函数

- 当派生类对基类的某个功能进行扩展时，他定义的成员函数名可能会和基类的成员函数名重复。在这种情况下，**派生类的函数会覆盖基类的函数。这称为重定义基类的成员函数。**
- 重定义的情况下，引用基类的同名函数必须使用作用域运算符，否则会由于派生类成员函数实际上调用了自身而引起无穷递归。

将派生类对象隐式转换为基类对象

- 具体可分为三种情况：
  - 将派生类对象赋值给基类对象
  - 基类指针指向派生类对象
  - 基类的对象引用派生类的对象

通过基类的对象、指针或引用来调用成员函数，执行的都是基类的成员函数。

- 总而言之，就是从派生类退化为基类。

注意不能反过来

- 不能将**基类对象**赋给**派生类对象**，除非在基类中定义了向派生类转换的类型转换函数（运算符重载函数）。
- 不能将**基类对象的地址**赋给**派生类指针**。
- 也不能将**基类指针**赋给**派生类指针**。如果程序员能确保这个基类指针指向的是一个派生类的对象，则可以用强制类型转换，表示程序员知道这个风险。

多态性

- 多态性的实现：分为静态联编和动态联编（联编：即绑定）
- 静态联编：在编译时根据当前对象的类型区分基类和派生类的同名成员函数，决定用哪一个函数实现某一功能。
- 动态联编：在运行时根据基类指针或引用指向的对象的实际类型，决定用哪一个函数实现某一功能。在C++中，使用虚函数来实现。

虚函数

- 虚函数的概念：在基类中用关键词virtual说明，并在派生类中重新定义的函数称为虚函数。

```
class Shape {
   public:  virtual void printShapeName() {cout<<“Shape”<<endl;}
};
```

- 在派生类中重定义虚函数时，其函数原型，包括返回类型、函数名、参数个数与参数类型的顺序都必须与基类中的原型完全相同。
- 当把一个函数定义为虚函数时，等于告诉编译器，这个成员函数在派生类中可能有不同的实现。必须在执行时根据实例对象的类型来决定调用基类还是派生类的函数。

虚函数的使用

- 当基类指针指向派生类对象，或基类对象引用派生类对象时，对基类指针或引用调用虚函数，系统会到派生类中寻找该函数的重定义。
  - 如找到则执行派生类中的虚函数，否则执行基类的虚函数。
  - 不是所有派生类都必须重定义基类的虚函数。
- 派生类在对基类的虚函数重定义时，关键字virtual可以写也可以不写。不管virtual写或者不写，该函数都自动成为虚函数。但最好是在重定义时写上virtual，这样便于理解。

虚析构函数

- 构造函数不能是虚函数。
- 析构函数可以是虚函数，而且最好是虚函数。
- 如果派生类新增的数据成员中有指针，指向堆内存，且派生类对象是通过基类指针操作的，那么 delete 基类指针 会调用基类的虚析构函数，然后又会找到并执行派生类的析构函数，避免内存泄漏。
- 如果继承层次树中的根类的析构函数是虚函数的话，所有派生类的析构函数都将是虚函数。

纯虚函数与抽象类

- 纯虚函数的一般形式（没有函数体，只有声明没有定义）：

  virtual 类型 函数名（参数表）= 0

- 如果一个类中至少有一个纯虚函数，则该类被称为抽象类。

- 抽象类的使用限制：
  - 抽象类只能作为其他类的基类，不能建立抽象类的对象。
  - 抽象类不能用作参数类型、函数返回类型或显式转换类型。
  - 如果派生类中定义了基类的所有纯虚函数，则该派生类不再是抽象类，否则仍为抽象类。

- **可以建立抽象类的指针或引用，可指向或引用它的派生类的对象。**

第14章 输入输出与文件

流与标准库

输入输出缓冲

基于控制台的I/O

基于文件的I/O

基于字符串的I/O

流的概念及用途

- C++的I/O操作是以字节流的形式实现的。流实际上就是字节序列。
- C++提供了无格式I/O和格式化I/O两种操作。
  - 无格式I/O传输速度快，但使用麻烦，多用于二进制文件读写。
  - 格式化I/O按不同的类型对数据进行处理，但需要增加额外的处理时间，不适于处理大容量的数据传输。

头文件	类型
iostream	istream从流中读取 ostream写到流中去 iostream对流进行读写，从istream和ostream派生
fstream	ifstream从文件中读取，由istream派生而来 ofstream写到文件中去，由ostream派生而来 fstream对流进行读写，由iostream派生而来

- 由于继承关系，所以cin、cout的运算符和成员函数也能被文件流对象使用。
  - 如 >>、<<、get、put、getline、read、write等等

输出缓冲区的刷新

- 取出缓冲区的内容后再将其清空的操作称为缓冲区的刷新。
  - 例如，刷新cout的缓冲区时，从其中取出的内容将显示在控制台上。
- 程序正常结束时，作为main函数返回工作的一部分，会刷新缓冲区。
- 当缓冲区已满时，在放入下一个值之前，会刷新缓冲区。
- 用标准库的操纵符（如行结束符endl）可显式地刷新缓冲区。
  - 这是 endl 和 '\n' 的又一个区别。
- 可将输出流与输入流关联起来。在这种情况下，在读输入流时，将刷新其关联的输出缓冲区。在标准库中，cout和cin被关联在一起，因此每个使用cin的输入操作都将刷新cout关联的缓冲区。

指针输出的特例

如果输出的是字符指针，C++并不输出该指针中保存的地址，而是输出该指针指向的字符串。如果确实想输出这个地址，可以用强制类型转换，将该指针转换成void*类型。

```
#include <iostream>
using namespace std;
int main() {
    char *ptr = "abcdef";
    cout << "ptr指向的内容为： " << ptr << endl;
    cout << "ptr中保存的地址为：" << (void*) ptr << endl;
    return 0;
}
```

ptr指向的内容为：
abcdef
ptr中保存的地址为：
0x0046C04C

流读取运算符>>

- 流读取运算符通常会跳过输入流中的空格、tab键、换行符等空白字符。
- 流读取运算符在读取成功时返回当前对象（如cin）的引用。
- 当遇到输入流中的文件结束符（EOF）时，流读取运算符返回0（EOF在各个系统中有不同表示，在Windows中是Ctrl+z）。
- 流读取运算符在读入EOF时返回0的特性使得它经常被用作为循环的判别条件，以避免选择特定的表示输入结束的值。

get函数

- 不带参数的get函数从当前对象读入一个字符，包括空白字符以及表示文件结束的EOF，并将读入的字符作为函数的返回值。如下列语句将输入的字符输出在显示器上，直到输入EOF：

```
while( (ch = cin.get()) != EOF ) cout<< ch;
```

- 带一个参数的get函数，它将输入流中的下一字符（包括空白字符）存储在参数中，它的返回值是当前对象的引用。如下列语句将输入一个字符串，存入字符数组ch，直到输入回车：

```
cin.get(ch[0]); for (i = 0; ch[i] != '\n'; ++i) cin.get(ch[i+1]); ch[i] = '\0';
```

带三个参数的get函数 vs getline函数

- 这两个函数都有三个参数，且都分别是字符数组的起始地址、读取的字符个数和分隔符(默认值为‘\n’)。
- 这两个函数都可以在读取比指定的字符个数**少一个字符**后结束，也可以在遇到分隔符时结束。读取结束后会在末尾添加‘\0’。
- 区别在于，get函数会把分隔符保留在输入流中，而getline函数会将分隔符删除。

格式化输入/输出

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

int main()
{
    int n;
    cout << "Enter an octal number: ";
    cin >> oct >> n;
    cout << "octal " << oct << n << " in hexdecimal is:" << hex << n << '\n' ;
    cout << "hexdecimal " << n << " in decimal is:" << dec << n << '\n' ;
    cout << setbase(8) << "octal " << n <<" in octal is:" << n << endl;
    return 0;
}
```

```
Enter an octal number: 30
Octal 30 in hexdecimal is: 18
Hexdecimal 18 in decimal is: 24
Octal 30 in octal is: 30
```

格式化输入/输出

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

int main() {
    double x = 123.456789, y = 9876.54321;

    for (int i = 9; i > 0; --i) { cout.precision(i); cout << x << '\t' << y << endl; }
    // 或写成：
    // for (int i = 9; i > 0; --i) cout << setprecision(i) << x << '\t' << y << endl;

    return 0;
}
```

格式化输入/输出

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

int main() {
    double x = 123.456789, y = 9876.54321;

    cout << fixed << setprecision(2) << x << '\t' << y << endl;
    cout << fixed << setprecision(3) << x << '\t' << y << endl;
    //加上fixed之后，setprecision的参数表示小数点后取几位

    return 0;
}
```

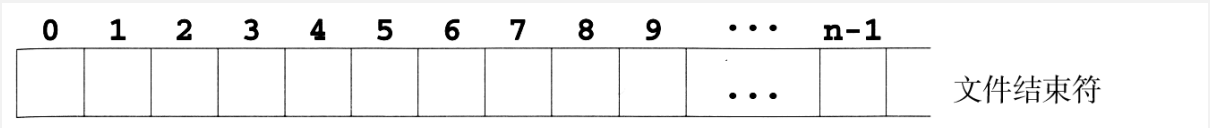
设置域宽

- 域宽是指数据所占的字符个数。
- 设置域宽可以用基类的成员函数width，也可以用流操纵符setw。width和setw都包含一个整型的参数，表示域宽。
- 设置域宽可用于输入，也可用于输出。**设置宽度仅适合于下一次输入或输出**，之后的操作的宽度将被设置为默认值。
- 未设置输出宽度时，按实际长度输出。如整型变量a=123，b=456，则：

```
cout << a << b; //将输出123456。
```

文件和流

- C++语言把每个文件都看成一个有序的字节流（把文件看成n个字节）。
- 每一个文件都以文件结束符(EOF)结束。



- ASCII文件是可以直接阅读的，而二进制文件则不行。
- ASCII文件中数据项占用的字节数量受数据值的影响。
  所以，一般不使用定位指针来随机访问ASCII文件中的数据。

打开和关闭文件

- 可以用流对象的成员函数open打开文件，或用流对象的构造函数打开文件。
  - 这两个函数都有两个参数：打开的文件名（字符串）和文件打开模式。
  - 如果文件打开失败，返回0。
- 当打开一个文件时，该文件就和某个流对象关联起来。
  - 流对象提供程序与特定文件或设备之间的通信通道。
- 文件访问完毕后，用成员函数close来关闭文件，切断流与文件的关联。
- 文件流对象包括ifstream、ofstream和fstream类的对象。

文件的顺序访问

- ASCII文件的读写和控制台读写一样，可以用流提取运算符“>>”从文件读数据，或用流插入运算符”<<”将数据写入文件，也可以用文件流的其他成员函数读写文件，如get函数，put函数等。
- 在读文件操作中，经常需要判断文件是否结束（数据是否被读完）。
- 可以利用基类ios的成员函数eof，该函数不需要参数，返回一个整型值。当读操作遇到文件结束时，该函数返回1，否则返回0。
- 也可以利用流提取操作的返回值，当“>>”操作成功时，返回true。

文件的随机访问

- 文件定位指针：是一个long类型的数据 ，指出当前读写的位置。
- C++文件有两个定位指针：读指针和写指针
  - 当文件以输入方式(ios::in)打开时，读指针指向文件中的第一个字节。
  - 当文件以输出方式(ios::out)打开时，写指针指向文件中的第一个字节。
  - 当文件以添加方式(ios::app)打开时，写指针指向文件末尾。
- 设置文件定位指针的成员函数：seekg（读）和seekp（写）。
  - 第一个参数通常为long类型的整数，表示偏移量。
  - 第二个参数说明该偏移量相对于哪个位置，包括ios::beg/cur/end。

访问与存储要求

- 可以立即访问到文件甚至大型文件中指定的记录。
- 可以在不破坏其他数据的情况下把数据添加到随机访问文件中。
- 可以在不重写整个文件的情况下更新以前存储的数据。

更新前：1 2 3 4 5 6 7 8 9 10　　10个数据

更新后：1 2 3 4 5 20 7 8 9 10　　9个数据，20和7变成了207

- 因此，要求记录的长度是固定的，即取决于数据类型而非数值。

write的无格式输出

- 调用成员函数write可实现无格式输出。
- 它的第一个参数是一个指向字符的指针，第二个参数是一个整型值，表示把一定数量的字节从字符数组中输出。例如对ofstream类对象fout:

- char buffer[ ] =“HAPPY BIRTHDAY”; //数组buffer的长度由字符串决定
  fout.write(buffer, 10); //将buffer中的前10个字节写入文件
- fout.write(reinterpret_cast<char *> (&x)), sizeof(x));
  // 将变量x在内存中的所有字节以二进制的形式写入文件，无视x的类型

read的无格式输入

- 调用成员函数read可实现无格式输入。
- 它的第一个参数是一个指向字符的指针，第二个参数是一个整型值，表示把一定数量的字节读入字符数组。例如对于ifstream类对象fin:

```
char buffer[80];  fin.read( buffer, 10 ); //从文件中读取10个字节，放入buffer
fin.read( reinterpret_cast<char *> (&x)), sizeof(x) );
// 从文件中读取适量字节以恰好填满变量x的内存空间，无视x的类型
```

- 如果还没有读到指定的字节个数就遇到了EOF，则读操作结束。此时可以用成员函数gcount统计读取的字节个数。

期末考试范围（2024秋）

不考的章节

- 2.3.3 输入异常
- 5.1.4 尾置返回类型
- 5.7 常量表达式函数
- 5.9 函数模板
- 6.5.2 string类
- 6.6 基于递归的算法
- 7.7.2 main函数的参数
- 7.8 函数指针
- 13 泛型机制——模板
- 14.5 基于字符串的输入输出
- 15 异常处理
- 16 容器和迭代器