网易一面：

多态性都有哪些？

（静态和动态，然后分别叙述一下虚函数和函数重载）

**多态**：指相同的对象收到不同的消息或者不同的对象收到相同的消息时产生的不同的实现动作。

**C++支持两种多态**：编译时多态（静态）、运行时多态（动态）

**编译时多态**：就是在编译期确定的一种多态。在C++中主要体现在函数模板，这里需要注意，函数重载和多态无关，很多地方把函数重载误认为是编译多态，这是错误的

1. #include <iostream>
2. **using** **namespace** std;
4. **template** <**typename** T>
5. T add(T a, T b)
6. {
7. t c = a+b;
8. **return** c;
9. }
11. **int** main()
12. {
13. **int** a1 = 1;
14. **int** b1 = 2;
15. **int** c1 = add(a1,b1);
16. cout<<"c1:"<<c1<<endl;
18. **double** a2 = 2.0;
19. **double** b2 = 4.0;
20. **double** c2 = add(a2,b2);
21. cout<<"c2:"<<c2<<endl;
22. }

在例1中，我们定义了一个函数模板，用来计算两个数的和。这两个数的数据类型在使用时才知道，main函数中调用同一个函数分别计算了两个int值和两个double值的和，这就体现了多态，在编译期，编译器根据一定的最佳匹配算法来确定函数模板的参数类型到底是什么，这就体现了编译期的多态性。

**运行时多态性**

     C++运行时多态性主要是通过虚函数来实现的。体现在具有继承关系的父类和子类之间，子类重新定义父类的成员函数称为覆盖或者重写，而虚函数允许子类重新定义父类的成员函数，即重写父类的成员函数。

     下面举例说明一下：

1. #include <iostream>
2. **using** **namespace** std;
4. **class** A{
5. **public**:
6. **void** f1()
7. {
8. cout<<"A::f1()"<<endl;
9. }
10. **virtual** **void** f2()
11. {
12. cout<<"A::f2()"<<endl;
13. }
14. };
16. **class** B:**public** A
17. {
18. **public**:
19. //覆盖
20. **void** f1()
21. {
22. cout<<"B::f1()"<<endl;
23. }
24. //重写
25. **virtual** **void** f2()
26. {
27. cout<<"B::f2()"<<endl;
28. }
29. };
31. **int** main()
32. {
33. A\* p = **new** B();
34. B\* q = **new** B();
35. p->f1();          //调用A::f1()
36. p->f2();          //调用B::f2()，体现多态性
37. q->f1();          //调用B::f1()
38. q->f2();          //调用B::f2()
39. **return** 0;
40. }

说说例2中关于体现多态性的问题，我们在父类即A类中定义了一个虚函数f2()——由关键字virtual修饰。既然是虚函数，允许子类重写这个函数，于是我们在子类即B类中重写了函数f2()。之后我们在main函数中定义了一个A类指针p，请注意，虽然定义的是一个父类指针，但是它指向的却是一个子类的对象（new B())，然后我们用这个父类的指针调用f2()，从结果来看，实际上调用的是子类的f2()，并不是父类的f2()，这就体现了多态性。虽然p是父类指针，但是它指向的是子类对象，而且调用的又是虚函数，那么在运行期，就会找到动态绑定到父类指针上的子类对象，然后查找子类的虚函数表，找到函数f2()的入口地址，从而调用子类的f2()函数，这就是运行期多态。

接下来我们再来看看p->f1();这句话，从运行结果来看，调用的是父类的f1()函数，这里是为什么没有体现多态呢？原因很简单，因为f1()不是虚函数，所以根本就没有多态性，虽然子类和父类都有f1()函数，但是子类仅仅是覆盖或者说是隐藏了父类的f1()函数，注意这里不是重写，是覆盖。而p是父类指针，所以只能调用父类的f1()函数。而指针q的类型为子类指针，所以q->f1();调用子类的函数f1()，q->f2();调用子类的函数f2();

**C++纯虚函数**

定义：纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法，基类中实现纯虚函数的方法是在函数原型后面加“=0”。例如：

1. **virtual** **void** f() = 0;

为什么要引入纯虚函数：

1、为了使用多态特性，我们必须在基类中定义虚拟函数。

2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。 为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。纯虚函数永远不会被调用，它们主要用来统一管理子类对象。

动态绑定怎么实现？

（问一下基类与派生类指针和引用的转换问题）

1.为每一个包含虚函数的类设置一个虚表（VTABLE）每当创建一个包含有虚函数的类或从包含虚函数的类派生一个类时，编译器就会为这个类创建一个VTABLE。在VTABLE中，编译器放置了这个类中，或者它的基类中所有已经声明为 virtual的函数的地址。如果在这个派生类中没有对基类中声明为 virtual 的函数进行重新定义，编译器就使用基类的这个虚函数的地址。而且所有VTABLE中虚函数地址的顺序是完全相同的。

2.初始化虚指针（VPTR）然后编译器在这个类的各个对象中放置VPTR。VPTR在对象的相同的位置（通常都在对象的开头）。VPTR必须被初始化为指向相应的VTABLE。

3.为虚函数调用插入代码 当通过基类的指针调用派生类的虚函数时，编译器将在调用处插入相应的代码，以实现通过VPTR找到VTABLE，并根据VTABLE中存储的正确的虚函数地址，访问到正确的函数。

为了支持c++的多态性，才用了动态绑定和静态绑定。理解他们的区别有助于更好的理解多态性，以及在编程的过程中避免犯错误。  
需要理解四个名词：  
1、对象的静态类型：对象在声明时采用的类型。是在编译期确定的。  
2、对象的动态类型：目前所指对象的类型。是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。  
关于对象的静态类型和动态类型，看一个示例：

1. **class** B
2. {
3. }
4. **class** C : **public** B
5. {
6. }
7. **class** D : **public** B
8. {
9. }
10. D\* pD = **new** D();//pD的静态类型是它声明的类型D\*，动态类型也是D\*
11. B\* pB = pD;//pB的静态类型是它声明的类型B\*，动态类型是pB所指向的对象pD的类型D\*
12. C\* pC = **new** C();
13. pB = pC;//pB的动态类型是可以更改的，现在它的动态类型是C\*

3、静态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。  
4、动态绑定：绑定的是对象的动态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。

**[cpp]** [view plain](http://blog.csdn.net/chgaowei/article/details/6427731) [copy](http://blog.csdn.net/chgaowei/article/details/6427731)

1. **class** B
2. {
3. **void** DoSomething();
4. **virtual** **void** vfun();
5. }
6. **class** C : **public** B
7. {
8. **void** DoSomething();//首先说明一下，这个子类重新定义了父类的no-virtual函数，这是一个不好的设计，会导致名称遮掩；这里只是为了说明动态绑定和静态绑定才这样使用。
9. **virtual** **void** vfun();
10. }
11. **class** D : **public** B
12. {
13. **void** DoSomething();
14. **virtual** **void** vfun();
15. }
16. D\* pD = **new** D();
17. B\* pB = pD;

让我们看一下，pD->DoSomething()和pB->DoSomething()调用的是同一个函数吗？  
不是的，虽然pD和pB都指向同一个对象。因为函数DoSomething是一个no-virtual函数，它是静态绑定的，也就是编译器会在编译期根据对象的静态类型来选择函数。pD的静态类型是D\*，那么编译器在处理pD->DoSomething()的时候会将它指向D::DoSomething()。同理，pB的静态类型是B\*，那pB->DoSomething()调用的就是B::DoSomething()。

让我们再来看一下，pD->vfun()和pB->vfun()调用的是同一个函数吗？  
是的。因为vfun是一个虚函数，它动态绑定的，也就是说它绑定的是对象的动态类型，pB和pD虽然静态类型不同，但是他们同时指向一个对象，他们的动态类型是相同的，都是D\*，所以，他们的调用的是同一个函数：D::vfun()。

上面都是针对对象指针的情况，对于引用（reference）的情况同样适用。

指针和引用的动态类型和静态类型可能会不一致，但是对象的动态类型和静态类型是一致的。

D D;  
D.DoSomething()和D.vfun()永远调用的都是D::DoSomething()和D::vfun()。

至于那些是动态绑定，那些是静态绑定，有篇文章总结的非常好:  
我总结了一句话：只有虚函数才使用的是动态绑定，其他的全部是静态绑定。目前我还没有发现不适用这句话的，如果有错误，希望你可以指出来。

特别需要注意的地方  
当缺省参数和虚函数一起出现的时候情况有点复杂，极易出错。我们知道，虚函数是动态绑定的，但是为了执行效率，缺省参数是静态绑定的。

1. **class** B
2. {
3. **virtual** **void** vfun(**int** i = 10);
4. }
5. **class** D : **public** B
6. {
7. **virtual** **void** vfun(**int** i = 20);
8. }
9. D\* pD = **new** D();
10. B\* pB = pD;
11. pD->vfun();
12. pB->vfun();

有上面的分析可知pD->vfun()和pB->vfun()调用都是函数D::vfun()，但是他们的缺省参数是多少？  
分析一下，缺省参数是静态绑定的，pD->vfun()时，pD的静态类型是D\*，所以它的缺省参数应该是20；同理，pB->vfun()的缺省参数应该是10。编写代码验证了一下，正确。  
对于这个特性，估计没有人会喜欢。所以，永远记住：  
“绝不重新定义继承而来的缺省参数（Never redefine function’s inherited default parameters value.）”

关于c++语言  
目前我基本上都是在c++的子集“面向对象编程”下工作，对于更复杂的知识了解的还不是很多。即便如此，到目前为止编程时需要注意的东西已经很多，而且后面可能还会继续增多，这也许是很多人反对c++的原因。  
c++是Google的四大官方语言之一。但是Google近几年确推出了go语言，而且定位是和c/c++相似。考虑这种情况，我认为可能是Google的程序员们深感c++的复杂，所以想开发一种c++的替代语言。有时间要了解一下go语言，看它在类似c++的问题上时如何取舍的。

类型转换有哪些？

（四种类型转换，分别举例说明）

**1、  static\_cast:**

功能：完成编译器认可的隐式类型转换。

格式type1 a;

type2 b = staic\_cast<type1>(a);将type1的类型转化为type2的类型；

使用范围：

（1）基本数据类型之间的转换，如int->double；

int a = 6;

double b = static\_cast<int>(a);

（2）派生体系中向上转型：将派生类指针或引用转化为基类指针或引用（向上转型）；

class base{       ….     }

class derived : public base{      ….     }

base \*b;

derived \*d = new derived();

b = static\_cast<base \*>(d);

**2、  dynamic\_cast**

功能：执行派生类指针或引用与基类指针或引用之间的转换。

格式：

（1）      其他三种都是编译时完成的，dynamic\_cast是运行时处理的，运行时要进行运行时类型检查；

（2）      基类中要有虚函数，因为运行时类型检查的类型信息在虚函数表中，有虚函数才会有虚函数表；

（3）      可以实现向上转型和向下转型，前提是必须使用public或protected继承；

例子：

向上转型：

class base{       …      };

class derived : public base{      …      };

int main()

{

base \*pb;

derived \*pd = new derived();

pb = dynamic\_cast<base \*>(pd);

return 0;

}

向下转型：

class base{       virtualvoid func(){}           };

class derived : public base{      void func(){}      };

int main()

{

base \*pb = new base();

derived \*pd = dynamic\_cast<derived \*>(pb);//向下转型

return 0;

}

**3、const\_cast:**

只能对指针或者引用去除或者添加const属性，对于变量直接类型不能使用const\_cast；不能用于不同类型之间的转换，只能改变同种类型的const属性。

如：

const int a= 0;

int b = const\_cast<int>(a);//不对的

const int \*pi = &a;

int \* pii = const\_cast<int \*>pi;//去除指针中的常量性，也可以添加指针的常量性;

const\_cast的用法：

（1）常用于函数的形参是一个非const的引用，我想要穿进去一个const的引用，可以使用const\_cast<Type&>para;去除实参的常量性，以便函数能够接受这个参数。

（2）一个const对象，我们想要调用该对象中的非const函数，可以使用const\_cast去除对象的常量性；

**4、reinterpret\_cast：**

从字面意思理解是一个“重新解释的类型转换”。也就是说对任意两个类型之间的变量我们都可以个使用reinterpret\_cast在他们之间相互转换，无视类型信息。

不常使用。

操作符重载（+操作符），具体如何去定义？

（让把操作符重载函数原型说一遍）

operator是C++的关键字，它和运算符一起使用，表示一个运算符函数，理解时应将operator=整体上视为一个函数名。

   这是C++扩展运算符功能的方法，虽然样子古怪，但也可以理解：一方面要使运算符的使用方法与其原来一致，另一方面扩展其功能只能通过函数的方式（c++中，“功能”都是由函数实现的)。

 一、为什么使用操作符重载？  
对于系统的所有操作符，一般情况下，只支持基本数据类型和标准库中提供的class，对于用户自己定义的class，如果想支持基本操作，比如比较大小，判断是否相等，等等，则需要用户自己来定义关于这个操作符的具体实现。比如，判断两个人是否一样大，我们默认的规则是按照其年龄来比较，所以，在设计person 这个class的时候，我们需要考虑操作符==，而且，根据刚才的分析，比较的依据应该是age。**那么为什么叫重载呢？**这是因为，在编译器实现的时候，已经为我们提供了这个操作符的基本数据类型实现版本，但是现在他的操作数变成了用户定义的数据类型class，所以，需要用户自己来提供该参数版本的实现。

二、如何声明一个重载的操作符？  
A:  操作符重载实现为类成员函数  
重载的操作符在类体中被声明，声明方式如同普通成员函数一样，只不过他的名字包含关键字operator，以及紧跟其后的一个c++预定义的操作符。  
可以用如下的方式来声明一个预定义的==操作符:  
class person{  
private:  
    int age;  
    public:  
    person(int a){  
       this->age=a;  
    }  
   inline bool operator == (const person &ps) const;  
};  
实现方式如下：  
inline bool person::operator==(const person &ps) const  
{

     if (this->age==ps.age)  
        return true;  
     return false;  
}  
调用方式如下：  
#include  
using namespace std;  
int main()  
{

  person p1(10);  
  person p2(20);  
  if(p1==p2) c

cout<<”the age is equal!”< return 0;  
}  
这里，因为operator ==是class person的一个成员函数，所以对象p1,p2都可以调用该函数，上面的if语句中，相当于p1调用函数==，把p2作为该函数的一个参数传递给该函数，从而实现了两个对象的比较。

B:操作符重载实现为非类成员函数(全局函数)  
对于全局重载操作符，代表左操作数的参数必须被显式指定。例如：  
#include  
#include  
using namespace std;  
class person  
{  
public:  
int age;  
public:  
};  
  
bool operator==(person const &p1 ,person const & p2)

//满足要求，做操作数的类型被显示指定  
{  
if(p1.age==p2.age)  
 return true;  
return false;  
}  
int main()  
{  
person rose;  
person jack;  
rose.age=18;  
jack.age=23;  
if(rose==jack)  
cout<<"ok"< return 0;  
}

C:**如何决定把一个操作符重载为类成员函数还是全局名字空间的成员呢**？  
①如果一个重载操作符是类成员，那么只有当与他一起使用的左操作数是该类的对象时，该操作符才会被调用。如果该操作符的左操作数必须是其他的类型，则操作符必须被重载为全局名字空间的成员。  
②C++要求赋值=，下标[]，调用()， 和成员指向-> 操作符必须被定义为类成员操作符。任何把这些操作符定义为名字空间成员的定义都会被标记为编译时刻错误。  
③如果有一个操作数是类类型如string类的情形那么对于对称操作符比如等于操作符最好定义为全局名字空间成员。

D:**重载操作符具有以下限制**：

1. 只有C++预定义的操作符集中的操作符才可以被重载；

C++允许重载的运算符  
C++中绝大部分运算符都是可以被重载的。

不能重载的运算符只有5个：

.             （成员访问运算符）

.\*            （成员指针访问运算符）

::             （域运算符）

sizeof    （长度运算符）

?:            （条件运算符）

前两个运算符不能重载是为了保证访问成员的功能不能被改变，域运算符合sizeof运算符的运算对象是类型而不是变量或一般表达式，不具备重载的特征。

（2）对于内置类型的操作符，它的预定义不能被改变，应不能为内置类型重载操作符，如，不能改变int型的操作符+的含义；

（3） 也不能为内置的数据类型定义其它的操作符；

（4） 只能重载类类型或枚举类型的操作符；

（5） 重载操作符不能改变它们的操作符优先级；

（6） 重载操作符不能改变操作数的个数；

（7） 除了对( )操作符外，对其他重载操作符提供缺省实参都是非法的；

实例1 重载operator()：

1. struct join\_if\_joinable
2. {
3. void operator()(thread& t)
4. {
5. if (t.joinable())
6. {
7. t.join();
8. }
9. }
10. };
11. //use
12. join\_if\_joinable(thread1);

内存对齐原则？

（原则叙述了一下并举例说明）

一、内存对齐的原因  
1、平台移植原因：不是所有的硬件平台都能任意访问任意地址上的数据，有些硬件平台只能在某些特定地址处读取特定的数据，否则会抛出硬件异常；  
2、性能原因：数据结构（尤其是栈）应尽可能的在自然边界对齐。原因在于，访问未对齐的内存，处理器需要进行两次访问，而访问对齐的内存，处理器只需要进行一次访问。  
二、内存对齐的规则  
在具体讲内存对齐的规则之前引入一个名词：对齐系数，也叫对齐模数，每个编译器都有自己默认的对齐系数，VC6.0默认为8。程序员可以根据需要进行修改，可通过预编译指令#pragma pack(n)，n就是对齐系数，可以取1、2、4、8、16，具体对齐规则有三条，如下：  
(1).数据成员的对齐规则：结构体(struct)(或者联合体(union))的数据成员，第一个数据成员放在偏移量为0的地方，以后每个数据成员按照#pragma pack(n)和数据成员中比较小的那个数对齐，也就是说，起始地址需要时这个数的倍数，具体下面会举例说明；  
(2).结构体(struct)(或者联合体(union))整体对齐规则：整体的大小应该按照#pragma pack(n)和结构中最长的数据结构中，最大的那个进行，也就是，需要是这个数的倍数；

(3).如果#pragma pack(n)比结构中任何一个数据成员类型都大，则对齐系数不起任何作用。下面举例说明，环境：VS2013，32位操作系统

**[objc]** [view plain](http://blog.csdn.net/captain_wangnb/article/details/50512598) [copy](http://blog.csdn.net/captain_wangnb/article/details/50512598)

1. #include <iostream>
2. **using** **namespace** std;
4. **struct** S
5. {
6. **char** a;
7. **int**  b;
8. **short** c;
9. };
11. **struct** T
12. {
13. **short** c;
14. **char**  a;
15. **int**   b;
16. };
18. **int** main()
19. {
20. cout << "sizeof(S) is " << **sizeof**(S) << endl;
21. cout << "sizeof(T) is " << **sizeof**(T) << endl;
22. system("PAUSE");
23. **return** 0;
24. }

代码输出结果为：

sizeof(S) is 12

sizeof(T) is 8

编译器默认对齐系数为8：

对于结构体S，成员a是char型数据，占1字节大小，b是int型数据，占用4个字节，因为规则1，所以需要从内存偏移量为4的倍数的地方开始，因此需要a需要补上3字节无用内存，此时a占用4字节，下面是c，c是short型数据，占用2个字节，此时偏移量4+4=8，是2的倍数，符合规则1，结构体此时总大小：4+4+2=10；又根据规则2，结构体成员变量类型，最长的为4，因此结构体整体大小应该为4的倍数，因此需要c多占用2个字节，此时结构体大小为：4+4+4=12。

对于结构体T，c是short型数据，占用2个字节大小，a是char型数据，占用1个字节大小，起始偏移量3，是1的倍数，满足，此时大小为：2+1=3，b为int型数据，占用4字节，起始偏移量为3，显然不满足规则1，需要b补充1个字节，此时起始偏移量为2+2=4，满足规则1，此时结构体T的大小为：2+2+4=8；又根据规则2，结构体成员变量类型中最长的为4，结构体内存大小满足，因此，最终大小为8。

模板怎么实现？

模板(Templates)是ANSI-C++ 标准中新引入的概念。如果你使用的 C++ 编译器不符合这个标准，则你很可能不能使用模板。

**函数模板( Function templates)**

模板(Templates)使得我们可以生成**通用的函数**，这些函数能够接受任意数据类型的参数，可返回任意类型的值，而不需要对所有可能的数据类型进行函数重载。这在一定程度上实现了宏（macro）的作用。它们的原型定义可以是下面两种中的任何一个：

template <class identifier> function\_declaration;  
template <typename identifier> function\_declaration;

上面两种原型定义的不同之处在关键字class 或 typename的使用。它们实际是完全等价的，因为两种表达的意思和执行都一模一样。

例如，要生成一个模板，返回两个对象中较大的一个，我们可以这样写：

template <class GenericType>

GenericType GetMax (GenericType a, GenericType b) { return (a>b?a:b); }

在第一行声明中，我们已经生成了一个通用数据类型的模板，叫做GenericType。因此在其后面的函数中，GenericType 成为一个有效的数据类型，它被用来定义了两个参数a和 b ，并被用作了函数GetMax的返回值类型。

GenericType 仍没有代表任何具体的数据类型；当函数 GetMax 被调用的时候，我们可以使用任何有效的数据类型来调用它。这个数据类型将被作为pattern来代替函数中GenericType 出现的地方。用一个类型pattern来调用一个模板的方法如下：

function <type> (parameters);

例如，要调用GetMax 来比较两个int类型的整数可以这样写：

int x,y;  
GetMax <int> (x,y);

因此，GetMax 的调用就好像所有的GenericType 出现的地方都用int 来代替一样。

这里是一个例子：

|  |  |
| --- | --- |
| // function template  #include <iostream.h>  template <class T> T GetMax (T a, T b) {      T result;      result = (a>b)? a : b;      return (result);  }  int main () {      int i=5, j=6, k;      long l=10, m=5, n;      k=GetMax(i,j);      n=GetMax(l,m);      cout << k << endl;      cout << n << endl;      return 0; } | **6 10** |

(在这个例子中，我们将通用数据类型命名为T 而不是 GenericType ，因为T短一些，并且它是模板更为通用的标示之一，虽然使用任何有效的标示符都是可以的。)

在上面的例子中，我们对同样的函数GetMax()使用了两种参数类型：int 和 long，而只写了一种函数的实现，也就是说我们写了一个函数的模板，用了两种不同的pattern来调用它。

如你所见，在我们的模板函数 GetMax() 里，类型 T 可以被用来声明新的对象

T result;

result 是一个T类型的对象， 就像a 和 b一样，也就是说，它们都是同一类型的，这种类型就是当我们调用模板函数时写在尖括号<> 中的类型。

在这个具体的例子中，通用类型 T 被用作函数GetMax 的参数，不需要说明<int>或 <long>，编译器也可以自动检测到传入的数据类型，因此，我们也可以这样写这个例子：

int i,j;  
GetMax (i,j);

因为i 和j 都是int 类型，编译器会自动假设我们想要函数按照int进行调用。这种暗示的方法更为有用，并产生同样的结果：

|  |  |
| --- | --- |
| // function template II  #include <iostream.h>  template <class T> T GetMax (T a, T b) {      return (a>b?a:b);  }  int main () {      int i=5, j=6, k;      long l=10, m=5, n;      k=GetMax(i,j);      n=GetMax(l,m);      cout << k << endl;      cout << n << endl;      return 0;  } | **6 10** |

注意在这个例子的main() 中我们如何调用模板函数GetMax() 而没有在括号<>中指明具体数据类型的。编译器自动决定每一个调用需要什么数据类型。

因为我们的模板函数只包括一种数据类型 (class T)， 而且它的两个参数都是同一种类型，我们不能够用两个不同类型的参数来调用它：

int i;  
long l;  
k = GetMax (i,l);

上面的调用就是不对的，因为我们的函数等待的是两个同种类型的参数。

我们也可以使得模板函数接受两种或两种以上类型的数据，例如：

template <class T>

T GetMin (T a, U b) { return (a<b?a:b); }

在这个例子中，我们的模板函数 GetMin() 接受两个不同类型的参数，并返回一个与第一个参数同类型的对象。在这种定义下，我们可以这样调用该函数：

int i,j;  
long l;  
i = GetMin <int, long> (j,l);

或者，简单的用

i = GetMin (j,l);

虽然 j 和 l 是不同的类型。

**类模板(Class templates)**

我们也可以定义类模板（class templates），使得一个类可以有基于通用类型的成员，而不需要在类生成的时候定义具体的数据类型，例如：

template <class T>

class pair {

    T values [2];

public:

    pair (T first, T second) {

        values[0]=first;

        values[1]=second;

    }

};

上面我们定义的类可以用来存储两个任意类型的元素。例如，如果我们想要定义该类的一个对象，用来存储两个整型数据115 和 36 ，我们可以这样写：

pair<int> myobject (115, 36);

我们同时可以用这个类来生成另一个对象用来存储任何其他类型数据，例如：

pair<float> myfloats (3.0, 2.18);

在上面的例子中，类的唯一一个成员函数已经被inline 定义。如果我们要在类之外定义它的一个成员函数，我们必须在每一函数前面加template <... >。

|  |  |
| --- | --- |
| // class templates  #include <iostream.h>  template <class T> class pair {      T value1, value2; public:     pair (T first, T second) {          value1=first;          value2=second;      }      T getmax ();  };  template <class T>  T pair::getmax (){      T retval;      retval = value1>value2? value1 : value2;      return retval;  }  int main () {      pair myobject (100, 75);      cout << myobject.getmax();      return 0;  } | **100** |

注意成员函数getmax 是怎样开始定义的：

template <class T>  
T pair::getmax ()

所有写 T 的地方都是必需的，每次你定义模板类的成员函数的时候都需要遵循类似的格式（这里第二个T表示函数返回值的类型，这个根据需要可能会有变化）。

**模板特殊化(Template specialization)**

模板的特殊化是当模板中的pattern有确定的类型时，模板有一个具体的实现。例如假设我们的类模板pair 包含一个取模计算（module operation）的函数，而我们希望这个函数只有当对象中存储的数据为整型(int)的时候才能工作，其他时候，我们需要这个函数总是返回0。这可以通过下面的代码来实现：

|  |  |
| --- | --- |
| // Template specialization  #include <iostream.h>  template <class T> class pair {      T value1, value2;  public:      pair (T first, T second){          value1=first;          value2=second;      }      T module () {return 0;}  };  template <>  class pair <int> {      int value1, value2;  public:      pair (int first, int second){          value1=first;          value2=second;      }      int module ();  };  template <>  int pair<int>::module() {      return value1%value2;  }  int main () {      pair <int> myints (100,75);      pair <float> myfloats (100.0,75.0);      cout << myints.module() << '\n';      cout << myfloats.module() << '\n';      return 0;  } | **25 0** |

由上面的代码可以看到，模板特殊化由以下格式定义：

template <> class class\_name <type>

这个特殊化本身也是模板定义的一部分，因此，我们必须在该定义开头写template <>。而且因为它确实为一个具体类型的特殊定义，通用数据类型在这里不能够使用，所以第一对尖括号<> 内必须为空。在类名称后面，我们必须将这个特殊化中使用的具体数据类型写在尖括号<>中。

当我们特殊化模板的一个数据类型的时候，同时还必须重新定义类的所有成员的特殊化实现（如果你仔细看上面的例子，会发现我们不得不在特殊化的定义中包含它自己的构造函数 constructor，虽然它与通用模板中的构造函数是一样的）。这样做的原因就是**特殊化不会继承通用模板的任何一个成员**。

**模板的参数值(Parameter values for templates)**

除了模板参数前面跟关键字class 或 typename 表示一个通用类型外，函数模板和类模板还可以包含其它不是代表一个类型的参数，例如代表一个常数，这些通常是基本数据类型的。例如，下面的例子定义了一个用来存储数组的类模板：

|  |  |
| --- | --- |
| // array template  #include <iostream.h>  template <class T, int N>  class array {      T memblock [N];  public:      void setmember (int x, T value);      T getmember (int x);  };  template <class T, int N>  void array<T,N>::setmember (int x, T value) {      memblock[x]=value;  }  template <class T, int N>  T array<T,N>::getmember (int x) {      return memblock[x];  }  int main () {      array <int,5> myints;      array <float,5> myfloats;      myints.setmember (0,100);      myfloats.setmember (3,3.1416);      cout << myints.getmember(0) << '\n';      cout << myfloats.getmember(3) << '\n';      return 0;  } | **100 3.1416** |

我们也可以为模板参数设置默认值，就像为函数参数设置默认值一样。

下面是一些模板定义的例子：

template <class T> // 最常用的：一个class 参数。

template <class T, class U> // 两个class 参数。

template <class T, int N> // 一个class 和一个整数。

template <class T = char> // 有一个默认值。

template <int Tfunc (int)> // 参数为一个函数。

**模板与多文件工程 (Templates and multiple-file projects)**

从编译器的角度来看，模板不同于一般的函数或类。它们在**需要时才被编译**(compiled on demand)，也就是说一个模板的代码直到需要生成一个对象的时候(instantiation)才被编译。当需要instantiation的时候，编译器根据模板为特定的调用数据类型生成一个特殊的函数。

当工程变得越来越大的时候，程序代码通常会被分割为多个源程序文件。在这种情况下，通常接口(interface)和实现(implementation)是分开的。用一个函数库做例子，接口通常包括所有能被调用的函数的原型定义。它们通常被定义在以.h 为扩展名的头文件 (header file) 中；而实现 (函数的定义) 则在独立的C++代码文件中。

模板这种类似宏(macro-like) 的功能，对多文件工程有一定的**限制：函数或类模板的实现 (定义) 必须与原型声明在同一个文件中**。也就是说我们不能再 将接口(interface)存储在单独的头文件中，而必须将接口和实现放在使用模板的同一个文件中。

回到函数库的例子，如果我们想要建立一个函数模板的库，我们不能再使用头文件(.h) ，取而代之，我们应该生成一个模板文件(template file)，将函数模板的接口和实现 都放在这个文件中 (这种文件没有惯用扩展名，除了**不要使用.h扩展名或不要不加任何扩展名**)。在一个工程中多次包含同时具有声明和实现的模板文件并不会产生链接错误 (linkage errors)，因为它们只有在需要时才被编译，而兼容模板的编译器应该已经考虑到这种情况，不会生成重复的代码。

指针和const的用法？

（就是四种情况说了一下）

1、**C++函数声明时在后面加const的作用**：非静态成员函数后面加const（加到非成员函数或静态成员后面会产生编译错误），表示成员函数隐含传入的this指针为 const指针，决定了在该成员函数中，任意修改它所在的的成员的操作都是不允许的（因为隐含了对this指针的const引用修饰）；唯一的例外是对于 mutable修饰的成员。加了const的成员函数可以被非const对象和const对象调用，但不加const的成员函数只能被非const对象调用。

2、**Cons修饰普通变量**

一般有两种写法：const int value;//即Value的值不能被改变

                          int const value;//即value的值不能被改变

上述的两种写法效果都是一样的。

3、**Const修饰指针类型变量**

A.const char\* pContent;//也可写成const （char）\* pContent；

B.char\* const pContent;//也可写成（char\*）const pContent；

C.char const\* pContent;//也可写成（char）Const \*pContent；

D.const char\* const pContent;

对上述的总结可以将A与C分为一类描述的结果都是const修饰\*pContent做指向的内容不能被改变，对于B来说也可以写成const (Char\*) pContent;其含义是指const所修饰的是pConten是一个指针变量就是一个常量，本身不容被改变。D表示指针变量和指针变量所指向的内容都不能被改变。

4、**const修饰函数参数**

例如：void function(const int Var);//表示const修饰的Var的值不能被改变

常常const修饰参数也用引用来提升执行效率如下所示：

例如：void Funcation（const int& Var）;

5、**const修饰函数的返回值**

例如：const int funcation();

其含义是const所修饰的返回值必须是常量含义基本上与const修饰普通变量或者指针基本相同。

6.**const修饰类对象、对象指针、对象引用**

const修饰类对象表示该对象为常量对象，其中的任何成员都不能被修改。对于对象指针和对象引用也是一样。

const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。

7、**const与define的区别**

1、编译器处理方式不同

A.define宏是在预处理阶段展开。

B.const常量是编译运行阶段使用。

(2) 类型和安全检查不同

A.define宏没有类型，不做任何类型检查，仅仅是展开。

B.const常量有具体的类型，在编译阶段会执行类型检查。

(3) 存储方式不同

A.define宏仅仅是展开，有多少地方使用，就展开多少次，不会分配内存。

B.const常量会在内存中分配(可以是堆中也可以是栈中)。

**虚函数、纯虚函数、虚函数与析构函数？**

（纯虚函数何如定义，为什么虚构函数要定义成虚函数）

一、虚函数

只有用virtual声明类的成员函数，使之成为虚函数，不能将类外的普通函数声明为虚函数。因为**虚函数的作用**是允许在派生类中对基类的虚函数重新定义。所以虚函数只能用于类的继承层次结构中。

     一个成员函数被声明为虚函数后，在同一类族中的类就不能再定义一个非virtual的但与该虚函数具有相同的参数（包括个数和类型）和函数返回值类型的同名函数。

       根据什么考虑是否把一个成员函数声明为虚函数？

       ①  看成员函数所在的类是否会作为基类

       ② 看成员函数在类的继承后有无可能被更改功能，如果希望更改其功能的，一般应该将它声明为虚函数。

       如果成员函数在类被继承后功能不需修改，或派生类用不到该函数，则不要把它声明为虚函数。不要仅仅考虑到作为基类而把类中的所有成员函数都声明为虚函数。

      应考虑对成员函数的调用是通过对象名还是通过基类指针或引用去访问，如果是通过基类指针或引用去访问的，则应当声明为虚函数。有时在定义虚函数时，并不定义其函数体，即纯虚函数。它的作用只是定义了一个虚函数名,具体功能留给派生类去添加。

      说明：使用虚函数，系统要有一定的空间开销。当一个类带有虚函数时，编译系统会为该类构造一个**虚函数表**（vtbl），它是一个指针数组，存放每个虚函数的入口地址。系统在进行动态关联的时间开销很少，提高了多态性的效率。

二、纯虚函数

        有时候，基类中的虚函数是为了派生类中的使用而声明定义的，其在基类中没有任何意义。此类函数我们叫做纯虚函数，不需要写成空函数的形式，只需要声明成：

      virtual 函数类型 函数名（形参表列）=0;

       注意：纯虚函数没有函数体；

      最后面的“=0“并不代表函数返回值为0，只是形式上的作用，告诉编译系统”这是纯虚函数”；

      这是一个声明语句，最后应有分号。

     纯虚函数只有函数的名字但不具备函数的功能，不能被调用。在派生类中对此函数提供定义后，才能具备函数的功能，可以被调用。

三、虚析构函数

**析构函数的作用**是在对象撤销之前把类的对象从内存中撤销。通常系统只会执行基类的析构函数，不执行派生类的析构函数。

      只需要把基类的析构函数声明为虚函数，即**虚析构函数**，这样当撤销基类对象的同时也撤销派生类的对象，这个过程是动态关联完成的。

      如果将基类的析构函数声明为虚函数时，由该基类所派生的所有派生类的析构函数都自动成为虚函数，即使派生类的析构函数与基类的析构函数名字不相同。

       最好把基类的析构函数声明为虚函数，这将使所有派生类的析构函数自动成为虚函数，如果程序中显式delete运算符删除一个对象，而操作对象用了指向派生类对象的基类指针，系统会调用相应类的析构函数。

        构造函数不能声明为虚函数。

例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | #include <iostream>  using namespace std;    class Animal  {  public:      Animal()      {          cout << "Animal::Animal() is called" << endl;      };      virtual ~Animal()      {          cout << "Animal::~Animal() is called" << endl;      }      virtual void eat()       {          cout << "Animal::eat() is called" << endl;       }      virtual void walk()      {          cout << "Animal::walk() is called" << endl;      }      /\* data \*/  };  class Dog : public Animal  {  public:      Dog(int w,int h)      {          cout << "Dog::Dog() is called" << endl;          this->weight=w;          this->height=h;      }      virtual ~Dog()      {          cout << "Dog::~Dog() is called" << endl;      }      int weight;      int height;      void eat()      {          cout<<"i eat meat"<<endl;      }      void walk()      {          cout<<"run"<<endl;      }      /\* data \*/  };   int main(int argc, char const \*argv[])  {      /\* code \*/      Animal \*ani= new Dog(12,23);      Dog \*dog=new Dog(23,34);      ani->eat();      ani->walk();      dog->eat();      dog->walk();      delete ani;      //delete dog;      return 0;  } |

　（C++ 析构函数一般定义为虚函数）如果基类中析构函数没有定义为虚函数，则delete ani的时候，仅仅调用了父类的析构函数，子类的没有调用，如果在父类和子类的构造函数中都有动态内存分配，那么就会存在内存泄漏的问题。一般析构函数最好都写成虚函数，尤其是父类。

内联函数

（讲了一下内联函数的优点以及和宏定义的区别）

1.  内联函数

在C++中我们通常定义以下函数来求两个整数的最大值：  
int max(int a, int b)  
{

 return a > b ? a : b;

}

为这么一个小的操作定义一个函数的**好处**有：

① **阅读和理解**函数 max 的调用，要比读一条等价的条件表达式并解释它的含义要容易得多

② 如果需要做任何**修改**，修改函数要比找出并修改每一处等价表达式容易得多

③ 使用函数可以确保**统一**的行为，每个测试都保证以相同的方式实现

④ 函数可以**重用**，不必为其他应用程序重写代码

虽然有这么多好处，但是写成函数有一个潜在的缺点：调用函数比求解等价表达式要慢得多。在大多数的机器上，调用函数都要做很多工作：调用前要先保存寄存器，并在返回时恢复，复制实参，程序还必须转向一个新位置执行

C++中支持内联函数，其目的是为了提高函数的执行效率，用关键字 inline 放在函数定义(注意是定义而非声明，下文继续讲到)的前面即可将函数指定为内联函数，内联函数通常就是将它在程序中的每个调用点上“内联地”展开，假设我们将 max 定义为内联函数：

复制代码 代码如下:

inline int max(int a, int b)  
{

 return a > b ? a : b;

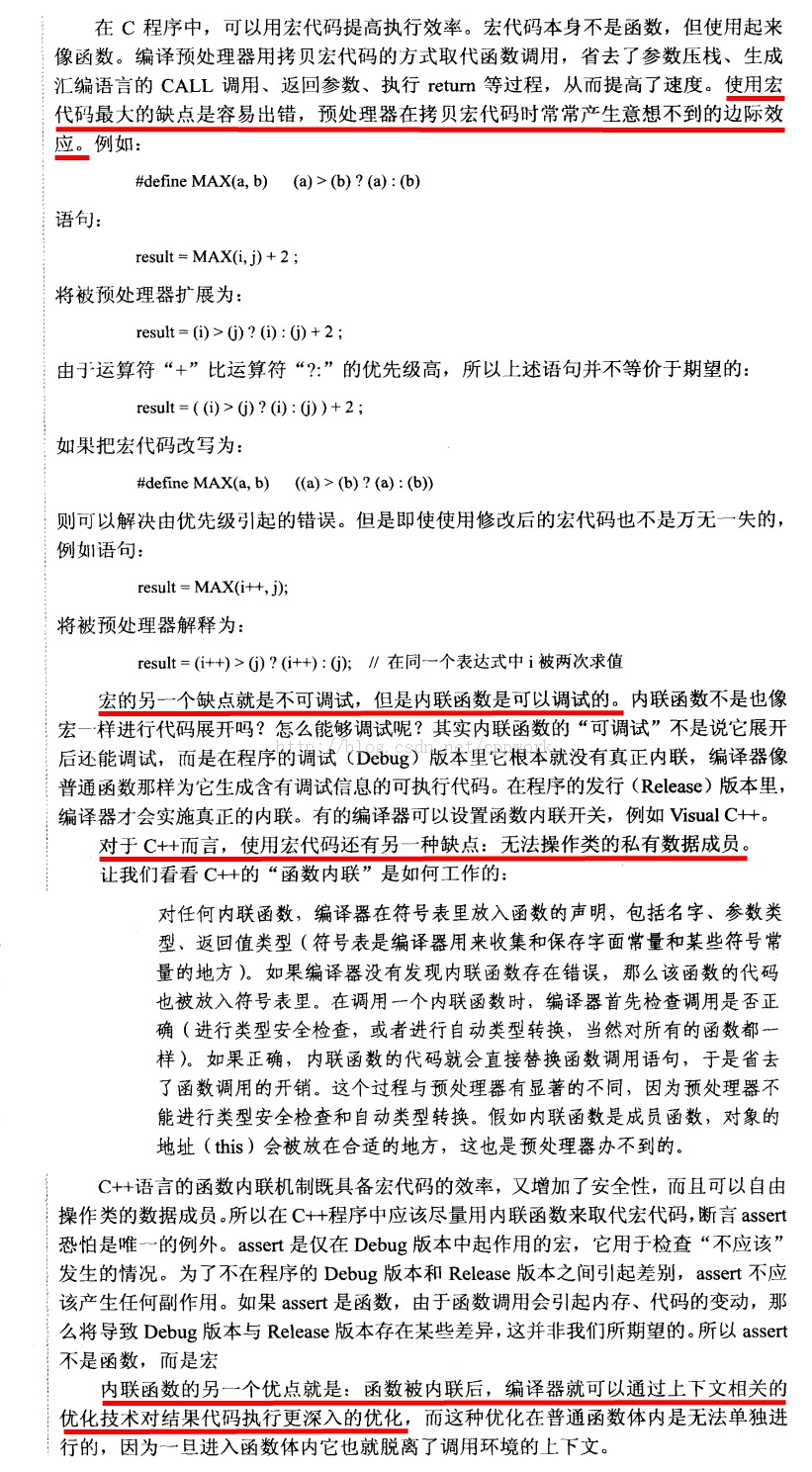
}

则调用： cout<<max(a, b)<<endl;  
在编译时展开为： cout<<(a > b ? a : b)<<endl;

从而**消除了把 max写成函数的额外执行开销**

2.  **内联函数和宏**

无论是《Effective C++》中的 “Prefer consts，enums，and inlines to #defines” 条款，还是《高质量程序设计指南——C++/C语言》中的“用函数内联取代宏”，宏在C++中基本是被废了，在书《高质量程序设计指南——C++/C语言》中这样解释到：

[](http://files.jb51.net/file_images/article/201401/2014126161246983.jpg)

3.  将内联函数放入头文件

关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。

如下风格的函数 Foo 不能成为内联函数：

inline void Foo(int x, int y);   // inline 仅与函数声明放在一起     
void Foo(int x, int y)  
{  
 ...  
}

而如下风格的函数 Foo 则成为内联函数：

void Foo(int x, int y);     
inline void Foo(int x, int y)   // inline 与函数定义体放在一起  
{  
 ...  
}

所以说，C++ inline函数是一种“**用于实现的关键字**”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了 inline 关键字，但我认为 inline 不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量C++/C 程序设计风格的一个基本原则：声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。

**定义在类声明之中的成员函数将自动地成为内联函数**，例如：

class A  
{

public:  
 void Foo(int x, int y) { ... }   // 自动地成为内联函数    
}

但是编译器是否将它真正内联则要看 Foo函数如何定义

**内联函数应该在头文件中定义**，这一点不同于其他函数。编译器在调用点内联展开函数的代码时，必须能够找到 inline 函数的定义才能将调用函数替换为函数代码，而对于在头文件中仅有函数声明是不够的。

**当然内联函数定义也可以放在源文件中，但此时只有定义的那个源文件可以用它**，而且必须为每个源文件拷贝一份定义(即每个源文件里的定义必须是完全相同的)，当然即使是放在头文件中，也是对每个定义做一份拷贝，只不过是编译器替你完成这种拷贝罢了。但相比于放在源文件中，放在头文件中既能够确保调用函数是定义是相同的，又能够保证在调用点能够找到函数定义从而完成内联(替换)。

但是你会很奇怪，重复定义那么多次，不会产生链接错误？

我们来看一个例子：

A.h :

class A  
{  
public:  
 A(int a, int b) : a(a),b(b){}  
 int max();

private:  
 int a;  
 int b;  
};

A.cpp :

#include "A.h"

inline int A::max()  
{

 return a > b ? a : b;  
}

Main.cpp :

#include <iostream>  
#include "A.h"  
using namespace std;

inline int A::max()  
{  
 return a > b ? a : b;  
}

int main()  
{  
 A a(3, 5);  
 cout<<a.max()<<endl;  
 return 0;  
}

一切正常编译，输出结果：5

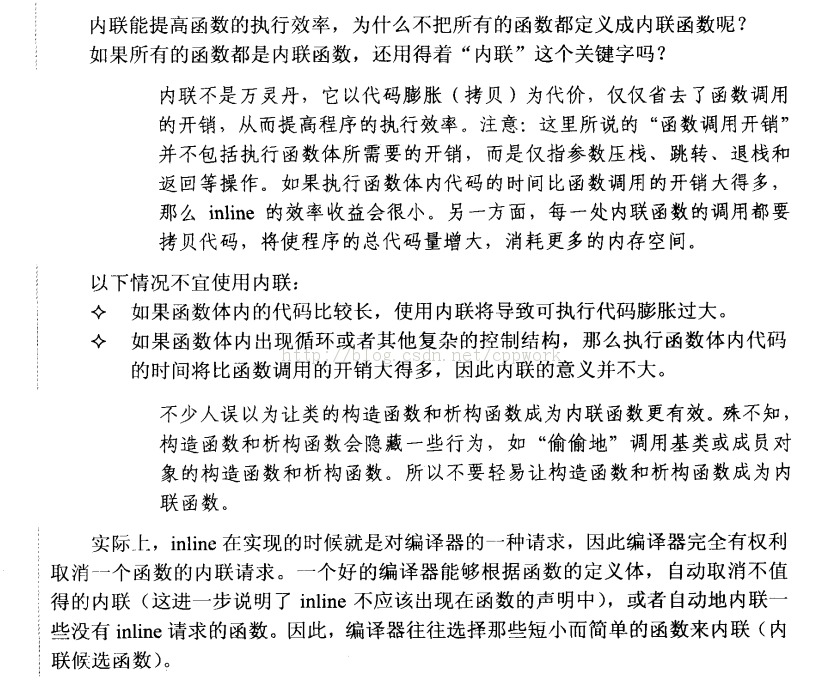
倘若你在Main.cpp中没有定义max内联函数，那么会出现链接错误：

error LNK2001: unresolved external symbol "public: int \_\_thiscall A::max(void)" (?max@A@@QAEHXZ)main.obj  
找不到函数的定义，所以**内联函数可以在程序中定义不止一次**，只要 inline 函数的定义在某个源文件中只出现一次，而且在所有源文件中，其定义必须是**完全相同**的就可以。

在头文件中加入或修改 inline 函数时，使用了该头文件的所有源文件都必须重新编译。

4.  慎用内联

内联虽有它的好处，但是也要慎用，以下摘自《高质量程序设计指南——C++/C语言》：

[](http://files.jb51.net/file_images/article/201401/2014126161607971.jpg)

而在Google C++编码规范中则规定得更加明确和详细：

内联函数：

Tip： **只有当函数只有 10 行甚至更少时才将其定义为内联函数.**

定义: 当函数被声明为内联函数之后, 编译器会将其内联展开, 而不是按通常的函数调用机制进行调用.  
优点: 当函数体比较小的时候, 内联该函数可以令目标代码更加高效. 对于存取函数以及其它**函数体比较短, 性能关键的函数**, 鼓励使用内联.  
缺点: 滥用内联将导致程序变慢. 内联可能使目标代码量或增或减, 这取决于内联函数的大小. 内联非常短小的存取函数通常会减少代码大小, 但内联一个相当大的函数将戏剧性的增加代码大小. 现代处理器由于更好的利用了指令缓存, 小巧的代码往往执行更快。  
结论: **一个较为合理的经验准则是, 不要内联超过 10 行的函数. 谨慎对待析构函数, 析构函数往往比其表面看起来要更长, 因为有隐含的成员和基类析构函数被调用!**另一个实用的经验准则: 内联那些包含循环或 switch 语句的函数常常是得不偿失 (除非在大多数情况下, 这些循环或 switch 语句从不被执行).  
有些函数即使声明为内联的也不一定会被编译器内联, 这点很重要; 比如**虚函数和递归函数就不会被正常内联**. 通常, 递归函数不应该声明成内联函数.(递归调用堆栈的展开并不像循环那么简单, 比如递归层数在编译时可能是未知的, 大多数编译器都不支持内联递归函数). 虚函数内联的主要原因则是想把它的函数体放在类定义内, 为了图个方便, 抑或是当作文档描述其行为, 比如精短的存取函数.

-inl.h文件：

Tip： 复杂的内联函数的定义, 应放在后缀名为 -inl.h 的头文件中.

**内联函数的定义必须放在头文件中, 编译器才能在调用点内联展开定义.** 然而, 实现代码理论上应该放在 .cc 文件中, 我们不希望 .h 文件中有太多实现代码, 除非在可读性和性能上有明显优势.

如果内联函数的定义比较短小, 逻辑比较简单, 实现代码放在 .h 文件里没有任何问题. 比如, 存取函数的实现理所当然都应该放在类定义内. 出于编写者和调用者的方便, 较复杂的内联函数也可以放到 .h 文件中, 如果你觉得这样会使头文件显得笨重, 也可以把它萃取到单独的 -inl.h 中. 这样把实现和类定义分离开来, 当需要时包含对应的 -inl.h 即可。

const和typedef

（主要将const的用处，有哪些优点）

**#DEFINE、CONST、TYPEDEF的差别(转)**

#define 并**不是定义变量**啊

**#define 只是用来做文本替换的**

例如：

#define Pi 3.1415926

float angel;

angel=30\*Pi/180;

那么，当程序进行编译的时候,编译器会首先将 “#define Pi 3.1415926”以后的，所有代码中的“Pi”全部换成 “3.1415926”

然后再进行编译。

我查到一个讲const与#define的差别的帖子，里面谈到const与#define最大的差别在于：前者在堆栈分配了**空间**，而后者只是把具体数值直接传递到目标变量罢了。或者说，const的常量是一个Run-Time的概念，他在程序中确确实实的存在并可以被调用、传递。而#define常量则是一个Compile-Time概念，它的生命周期止于编译期：在实际程序中他只是一个常数、一个命令中的参数，没有实际的存在。

**const常量存在于程序的数据段**，**#define常量存在于程序的代码段**。

至于两者的优缺点，要看具体的情况了。一般的常数应用，笔者个人认为#define是一个更好的选择：

i.从run-time的角度来看，他在空间上和时间上都有很好优势。

ii.从compile-time的角度来看，类似m=t\*10的代码不会被编译器优化，t\*10的操作需要在run-time执行。而#define的常量会被合并（在上例中T\*10将被0x82取代）。

但是：如果你需要粗鲁的修改常数的值，那就得使用const了，因为后者在程序中没有实际的存在。（其实应该说修改数据段比代码段要简单^\_^）。

**有关#define的用法**

**1.简单的define定义**#define MAXTIME 1000

程序中遇到MAXTIME就会当作1000来处理.  
一个简单的MAXTIME就定义好了，它代表1000，如果在程序里面写  
if(i<MAXTIME){.........}  
编译器在处理这个代码之前会对MAXTIME进行处理替换为1000。  
这样的定义看起来类似于普通的常量定义CONST，但也有着不同，因为define的定义更像是简单的文本替换，而不是作为一个量来使用，这个问题在下面反映的尤为突出。

**2.define的“函数定义”**  
define可以像函数那样接受一些参数，如下  
#define max(x,y) (x)>(y)?(x):(y);  
这个定义就将返回两个数中较大的那个，看到了吗？因为这个“函数”没有类型检查，就好像一个函数模板似的，当然，它绝对没有模板那么安全就是了。可以作为一个简单的模板来使用而已。  
但是这样做的话存在隐患，例子如下：  
#define Add(a,b) a+b;  
在一般使用的时候是没有问题的，但是如果遇到如：c \* Add(a,b) \* d的时候就会出现问题，代数式的本意是a+b然后去和c，d相乘，但是因为使用了define（它只是一个简单的替换），所以式子实际上变成了  
c\*a + b\*d  
另外举一个例子：  
#define pin (int\*);  
pin a,b;  
本意是a和b都是int型指针，但是实际上变成int\* a,b;  
a是int型指针，而b是int型变量。  
这时应该使用typedef来代替define，这样a和b就都是int型指针了。  
**所以我们在定义的时候，养成一个良好的习惯，建议所有的层次都要加括号。**  
**3.宏的单行定义**#define A(x) T\_##x  
#define B（x） #@x  
#define C（x） #x  
我们假设：x=1，则有：  
A(1)------）T\_1  
B(1)------）'1'  
C(1)------）"1"  
**4.define的多行定义**define可以替代多行的代码，例如MFC中的宏定义（非常的经典，虽然让人看了恶心）  
#define MACRO(arg1, arg2) do { \  
/\* declarations \*/ \  
stmt1; \  
stmt2; \  
/\* ... \*/ \  
} while(0) /\* (no trailing ; ) \*/  
关键是要在每一个换行的时候加上一个"\"   
摘抄自<http://www.blog.edu.cn/user1/16293/archives/2005/115370.shtml> 修补了几个bug  
**5.在大规模的开发过程中，特别是跨平台和系统的软件里，define最重要的功能是条件编译**就是：  
#ifdef WINDOWS  
......  
......  
#endif  
#ifdef LINUX  
......  
......  
#endif  
可以在编译的时候通过#define设置编译环境

**6.如何定义宏、取消宏**//定义宏  
#define [MacroName] [MacroValue]  
//取消宏  
#undef [MacroName]  
普通宏  
#define PI (3.1415926)  
带参数的宏  
#define max(a,b) ((a)>(b)? (a),(b))  
关键是十分容易产生错误，包括机器和人理解上的差异等等。  
**7.条件编译**  
#ifdef XXX…(#else) …#endif  
例如 #ifdef DV22\_AUX\_INPUT  
#define AUX\_MODE 3   
#else  
#define AUY\_MODE 3  
＃endif  
#ifndef XXX … (#else) … #endif  
**8.头文件(.h)可以被头文件或C文件包含**重复包含（重复定义）  
由于头文件包含可以嵌套，那么C文件就有可能包含多次同一个头文件，就可能出现重复定义的问题的。  
通过条件编译开关来避免重复包含（重复定义）  
例如  
#ifndef \_\_headerfileXXX\_\_  
＃define \_\_headerfileXXX\_\_  
…  
文件内容  
…  
#endif

typedef和#define的用法与区别

**一、typedef的用法**

在C/C++语言中，typedef常**用来定义一个标识符及关键字的别名**，它是语言编译过程的一部分，但它并**不实际分配内存空间**，实例像：

typedef    int       INT;  
typedef    int       ARRAY[10];  
typedef   (int\*)    pINT;

**typedef可以增强程序的可读性，以及标识符的灵活性，但它也有“非直观性”等缺点**。

**二、#define的用法**

#define为一宏定义语句，通常用它来定义常量(包括无参量与带参量)，以及用来实现那些“表面似和善、背后一长串”的宏，它本身并不在编译过程中进行，而是在这之前(预处理过程)就已经完成了，但也因此难以发现潜在的错误及其它代码维护问题，它的实例像：

#define   INT             int  
#define   TRUE         1  
#define   Add(a,b)     ((a)+(b));  
#define   Loop\_10    for (int i=0; i<10; i++)

在Scott Meyer的Effective C++一书的条款1中有关于#define语句弊端的分析，以及好的替代方法，大家可参看。

三、typedef与#define的区别

从以上的概念便也能基本清楚，typedef只是为了增加可读性而为标识符另起的新名称(仅仅只是个别名)，而#define原本在C中是为了定义常量，到了C++，const、enum、inline的出现使它也渐渐成为了起别名的工具。有时很容易搞不清楚与typedef两者到底该用哪个好，如#define INT int这样的语句，用typedef一样可以完成，用哪个好呢？我主张用typedef，因为在早期的许多C编译器中这条语句是非法的，只是现今的编译器又做了扩充。为了尽可能地兼容，一般都**遵循#define定义“可读”的常量以及一些宏语句的任务**，而**typedef则常用来定义关键字、冗长的类型的别名。**

**宏定义只是简单的字符串代换(原地扩展)**，而**typedef则不是原地扩展，它的新名字具有一定的封装性，以致于新命名的标识符具有更易定义变量的功能**。请看上面第一大点代码的第三行：

typedef    (int\*)      pINT**;**  
以及下面这行:  
#define    pINT2    int\*

效果相同？实则不同！实践中见差别：pINT a,b;的效果同int \*a; int \*b;表示定义了两个整型指针变量。而pINT2 a,b;的效果同int \*a, b;

表示定义了一个整型指针变量a和整型变量b。

注意：两者还有一个行尾;号的区别哦！

**const用法主要是防止定义的对象再次被修改,定义对象变量时要初始化变量**

下面我就介绍一下几种**常见的用法**

**1.用于定义常量变量,这样这个变量在后面就不可以再被修改**

 const int Val = 10;

 //Val = 20; //错误,不可被修改

**2. 保护传参时参数不被修改,**如果使用引用传递参数或按地址传递参数给一个函数,在这个函数里这个参数的值若被修改,则函数外部传进来的变量的值也发生改变,若想保护传进来的变量不被修改,可以使用const保护

 void  fun1(const int &val)

  {

     //val = 10; //出错

}

void fun2(int &val)

{

   val = 10; //没有出错

}

void main()

{

   int a = 2;

   int b = 2;

   fun1(a); //因为出错,这个函数结束时a的值还是2

   fun2(b);//因为没有出错,函数结束时b的值为10

}

**如果只想把值传给函数,而且这个不能被修改,则可以使用const保护变量,有人会问为什么不按值传递,按值传递还需要把这个值复制一遍,而引用不需要,使用引用是为了提高效率//如果按值传递的话,没必要加const,那样根本没意义**

**3. 节约内存空间**,

 #define  PI  3.14 //使用#define宏

 const double Pi = 3.14 //使用const,这时候Pi并没有放入内存中

 double  a = Pi;  //这时候才为Pi分配内存,不过后面再有这样的定义也不会再分配内存

 double  b = PI;  //编译时分配内存

 double  c = Pi;  //不会再分配内存,

 double  d = PI;  //编译时再分配内存

**const定义的变量,系统只为它分配一次内存,而使用#define定义的常量宏,能分配好多次,这样const就很节约空间**

**4.类中使用const修饰函数防止修改非static类成员变量**

 class

{

 public:

  void fun() const //加const修饰

   {

     a = 10; //出错,不可修改非static变量

     b = 10; //对,可以修改

}

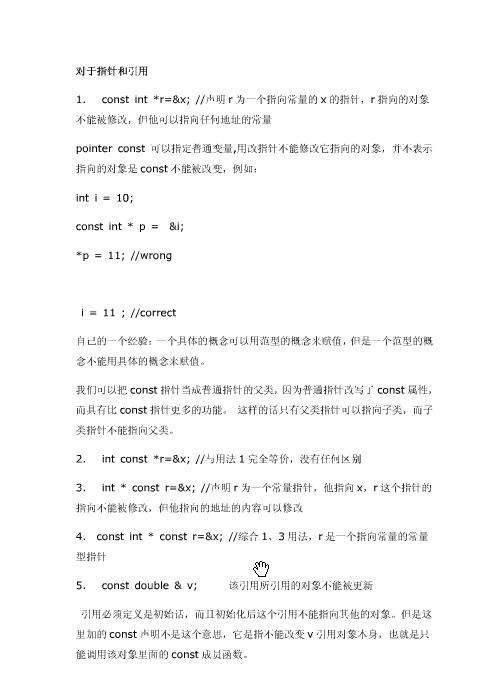
 private:

  int  a ;

  static int b;

}

**5.修饰指针**const int \*A; 或 int const \*A;  //const修饰指向的对象，A可变，A指向的对象不可变  
int \*const A; 　             //const修饰指针A， A不可变，A指向的对象可变   
const int \*const A;           //指针A和A指向的对象都不可变



**6.修饰函数返回值,防止返回值被改变**

  const int fun();

  接收返回值的变量也必须加const

**7.修饰类的成员变量**

  使用const修饰的变量必须初始化,在类中又不能在定义时初始化,

如;

class

{

private:

  int a = 10;

  const int b = 10;

  static const int c = 10;

//这样初始化都是错的,

}

**初始化const int类型(没有static),在类的构造函数上初始化**

Class Test

{

Public:

  Test():b(23) //构造函数上初始化b的值为23

   {

}

private:

     const int b ;

}

**初始化staticconst int这个类型的(带有static的),在类的外面初始化**

class Test

{

private:

  static const int c;

}

const int Test::c=10; //类的外部初始化c为10

8.const定义的对象变量只能作用于这个程序该C/C++文件,不能被该程序的其他C/C++文件调用,

 如file1.cpp中 const int val;

 在file2.cpp中, extern intval; //错误,无法调用,

**要想const定义的对象变量能被其他文件调用,定义时必须使用extern修饰为**

**extern const int val;**

**非const变量默认为extern,要是const能被其他文件访问必须显示指定为extern**

排序算法有哪些？快速排序怎么实现？最好时间复杂度，平均时间复杂度

### 比较排序目录

[冒泡排序](http://www.cnblogs.com/eniac12/p/5329396.html#s1)

[鸡尾酒排序](http://www.cnblogs.com/eniac12/p/5329396.html#s12)

[选择排序](http://www.cnblogs.com/eniac12/p/5329396.html#s2)

[插入排序](http://www.cnblogs.com/eniac12/p/5329396.html#s3)

[二分插入排序](http://www.cnblogs.com/eniac12/p/5329396.html#s31)

[希尔排序](http://www.cnblogs.com/eniac12/p/5329396.html#s32)

[归并排序](http://www.cnblogs.com/eniac12/p/5329396.html#s4)

[堆排序](http://www.cnblogs.com/eniac12/p/5329396.html#s5)

[快速排序](http://www.cnblogs.com/eniac12/p/5329396.html#s6)

　　我们通常所说的排序算法往往指的是**内部排序算法**，即数据记录在内存中进行排序。

　　排序算法大体可分为两种：

　　　　一种是**比较排序**，时间复杂度O(nlogn) ~ O(n^2)，主要有：**冒泡排序**，**选择排序**，**插入排序**，**归并排序**，**堆排序**，**快速排序**等。

　　　　另一种是**非比较排序**，时间复杂度可以达到O(n)，主要有：**计数排序**，**基数排序**，**桶排序**等。

　　这里我们来探讨一下常用的比较排序算法，非比较排序算法将在[下一篇文章](http://www.cnblogs.com/eniac12/p/5332117.html)中介绍。下表给出了常见比较排序算法的性能：



　　有一点我们很容易忽略的是**排序算法的稳定性**(腾讯校招2016笔试题曾考过)。

　　排序算法稳定性的简单形式化定义为：**如果Ai = Aj，排序前Ai在Aj之前，排序后Ai还在Aj之前，则称这种排序算法是稳定的。**通俗地讲就是保证排序前后两个相等的数的相对顺序不变。

　　对于不稳定的排序算法，只要举出一个实例，即可说明它的不稳定性；而对于稳定的排序算法，必须对算法进行分析从而得到稳定的特性。需要注意的是，**排序算法是否为稳定的是由具体算法决定的**，不稳定的算法在某种条件下可以变为稳定的算法，而稳定的算法在某种条件下也可以变为不稳定的算法。

　　例如，对于冒泡排序，原本是稳定的排序算法，如果将记录交换的条件改成A[i] >= A[i + 1]，则两个相等的记录就会交换位置，从而变成不稳定的排序算法。

　　其次，说一下排序算法稳定性的好处。**排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，前一个键排序的结果可以为后一个键排序所用。**基数排序就是这样，先按低位排序，逐次按高位排序，低位排序后元素的顺序在高位也相同时是不会改变的。

### 冒泡排序(Bubble Sort)

　　冒泡排序是一种极其简单的排序算法，也是我所学的第一个排序算法。它重复地走访过要排序的元素，依次比较相邻两个元素，如果他们的顺序错误就把他们调换过来，直到没有元素再需要交换，排序完成。这个算法的名字由来是因为越小(或越大)的元素会经由交换慢慢“浮”到数列的顶端。

　　冒泡排序算法的运作如下：

比较相邻的元素，如果前一个比后一个大，就把它们两个调换位置。

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

针对所有的元素重复以上的步骤，除了最后一个。

持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

　　由于它的简洁，冒泡排序通常被用来对于程序设计入门的学生介绍算法的概念。冒泡排序的代码如下：

[复制代码](javascript:void(0);)

#include <stdio.h>

// 分类 -------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- O(n^2)

// 最优时间复杂度 ---- 如果能在内部循环第一次运行时,使用一个旗标来表示有无需要交换的可能,可以把最优时间复杂度降低到O(n)

// 平均时间复杂度 ---- O(n^2)

// 所需辅助空间 ------ O(1)

// 稳定性 ------------ 稳定

void Swap(int A[], int i, int j)

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

void BubbleSort(int A[], int n)

{

for (int j = 0; j < n - 1; j++) // 每次最大元素就像气泡一样"浮"到数组的最后

{

for (int i = 0; i < n - 1 - j; i++) // 依次比较相邻的两个元素,使较大的那个向后移

{

if (**A[i] > A[i + 1]**) // 如果条件改成A[i] >= A[i + 1],则变为不稳定的排序算法

{

Swap(A, i, i + 1);

}

}

}

}

int main()

{

int A[] = { 6, 5, 3, 1, 8, 7, 2, 4 }; // 从小到大冒泡排序

int n = sizeof(A) / sizeof(int);

BubbleSort(A, n);

printf("冒泡排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

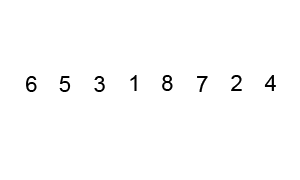
printf("\n");

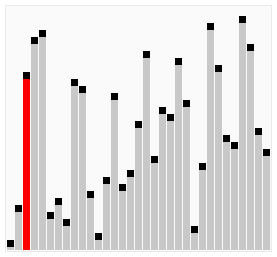
return 0;

}

[复制代码](javascript:void(0);)

　　上述代码对序列{ 6, 5, 3, 1, 8, 7, 2, 4 }进行冒泡排序的实现过程如下



　　使用冒泡排序为一列数字进行排序的过程如右图所示：　　

　　尽管冒泡排序是最容易了解和实现的排序算法之一，但它对于少数元素之外的数列排序是很没有效率的。

### 冒泡排序的改进：****鸡尾酒排序****

鸡尾酒排序，也叫**定向冒泡排序**，是冒泡排序的一种改进。此算法与冒泡排序的不同处在于**从低到高然后从高到低**，而冒泡排序则仅从低到高去比较序列里的每个元素。他可以得到比冒泡排序稍微好一点的效能。

　　鸡尾酒排序的代码如下：

[复制代码](javascript:void(0);)

#include <stdio.h>

// 分类 -------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- O(n^2)

// 最优时间复杂度 ---- 如果序列在一开始已经大部分排序过的话,会接近O(n)

// 平均时间复杂度 ---- O(n^2)

// 所需辅助空间 ------ O(1)

// 稳定性 ------------ **稳定**

void Swap(int A[], int i, int j)

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

void CocktailSort(int A[], int n)

{

int left = 0; // 初始化边界

int right = n - 1;

while (left < right)

{

for (int i = left; i < right; i++) // 前半轮,将最大元素放到后面

{

if (A[i] > A[i + 1])

{

Swap(A, i, i + 1);

}

}

right--;

for (int i = right; i > left; i--) // 后半轮,将最小元素放到前面

{

if (A[i - 1] > A[i])

{

Swap(A, i - 1, i);

}

}

left++;

}

}

int main()

{

int A[] = { 6, 5, 3, 1, 8, 7, 2, 4 }; // 从小到大定向冒泡排序

int n = sizeof(A) / sizeof(int);

CocktailSort(A, n);

printf("鸡尾酒排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

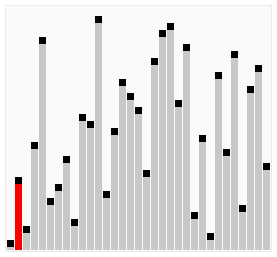
}

printf("\n");

return 0;

}

[复制代码](javascript:void(0);)

　　使用鸡尾酒排序为一列数字进行排序的过程如右图所示：　　

　　以序列(2,3,4,5,1)为例，鸡尾酒排序只需要访问一次序列就可以完成排序，但如果使用冒泡排序则需要四次。但是**在乱数序列的状态下，鸡尾酒排序与冒泡排序的效率都很差劲**。

### ****选择排序(Selection Sort)****

　　选择排序也是一种简单直观的排序算法。它的工作原理很容易理解：初始时在序列中找到最小（大）元素，放到序列的起始位置作为已排序序列；然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

　　注意**选择排序与冒泡排序的区别**：冒泡排序通过依次交换相邻两个顺序不合法的元素位置，从而将当前最小（大）元素放到合适的位置；而**选择排序每遍历一次都记住了当前最小（大）元素的位置**，最后仅需一次交换操作即可将其放到合适的位置。

　　选择排序的代码如下：

#include <stdio.h>

// 分类 -------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- O(n^2)

// 最优时间复杂度 ---- O(n^2)

// 平均时间复杂度 ---- O(n^2)

// 所需辅助空间 ------ O(1)

// 稳定性 ------------ 不稳定

void Swap(int A[], int i, int j)

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

void SelectionSort(int A[], int n)

{

for (int i = 0; i < n - 1; i++) // i为已排序序列的末尾

{

int min = i;

for (int j = i + 1; j < n; j++) // 未排序序列

{

if (A[j] < A[min]) // 找出未排序序列中的最小值

{

min = j;

}

}

if (min != i)

{

Swap(A, min, i); // 放到已排序序列的末尾，该操作很有可能把稳定性打乱，所以**选择排序是不稳定的排序算法**

}

}

}

int main()

{

int A[] = { 8, 5, 2, 6, 9, 3, 1, 4, 0, 7 }; // 从小到大选择排序

int n = sizeof(A) / sizeof(int);

SelectionSort(A, n);

printf("选择排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

printf("\n");

return 0;

}

[复制代码](javascript:void(0);)

 　　上述代码对序列{ 8, 5, 2, 6, 9, 3, 1, 4, 0, 7 }进行选择排序的实现

**选择排序是不稳定的排序算法，不稳定发生在最小元素与A[i]交换的时刻。**

比如序列：{ 5, 8, 5, 2, 9 }，一次选择的最小元素是2，然后把2和第一个5进行交换，从而改变了两个元素5的相对次序。

### 插入排序(Insertion Sort)

　　插入排序是一种简单直观的排序算法。它的工作原理非常类似于我们抓扑克牌



**对于未排序数据(右手抓到的牌)，在已排序序列(左手已经排好序的手牌)中从后向前扫描，找到相应位置并插入。**

　　插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

　　具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤2~5

　　插入排序的代码如下：

[复制代码](javascript:void(0);)

#include <stdio.h>

// 分类 ------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- 最坏情况为输入序列是降序排列的,此时时间复杂度O(n^2)

// 最优时间复杂度 ---- 最好情况为输入序列是升序排列的,此时时间复杂度O(n)

// 平均时间复杂度 ---- O(n^2)

// 所需辅助空间 ------ O(1)

// 稳定性 ------------ 稳定

void InsertionSort(int A[], int n)

{

for (int i = 1; i < n; i++) // 类似抓扑克牌排序

{

int get = A[i]; // 右手抓到一张扑克牌

int j = i - 1; // 拿在左手上的牌总是排序好的

while (j >= 0 && A[j] > get) // 将抓到的牌与手牌从右向左进行比较

{

A[j + 1] = A[j]; // 如果该手牌比抓到的牌大，就将其右移

j--;

}

A[j + 1] = get; // 直到该手牌比抓到的牌小(或二者相等)，将抓到的牌插入到该手牌右边(相等元素的相对次序未变，所以**插入排序是稳定的**)

}

}

int main()

{

int A[] = { 6, 5, 3, 1, 8, 7, 2, 4 };// 从小到大插入排序

int n = sizeof(A) / sizeof(int);

InsertionSort(A, n);

printf("插入排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

printf("\n");

return 0;

}

[复制代码](javascript:void(0);)

 　　上述代码对序列{ 6, 5, 3, 1, 8, 7, 2, 4 }进行插入排序的实现过程

**插入排序不适合对于数据量比较大的排序应用**。但是，如果需要排序的数据量很小，比如**量级小于千**，那么插入排序还是一个不错的选择。 插入排序在工业级库中也有着广泛的应用，在STL的sort算法和**stdlib的qsort**算法中，都将插入排序作为快速排序的补充，用于少量元素的排序（通常为8个或以下）。

### 插入排序的改进：二分插入排序

　　对于插入排序，如果比较操作的代价比交换操作大的话，可以采用**二分查找法**来减少比较操作的次数，我们称为**二分插入排序**，代码如下：

[复制代码](javascript:void(0);)

#include <stdio.h>

// 分类 -------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- O(n^2)

// 最优时间复杂度 ---- O(nlogn)

// 平均时间复杂度 ---- O(n^2)

// 所需辅助空间 ------ O(1)

// 稳定性 ------------ 稳定

void InsertionSortDichotomy(int A[], int n)

{

for (int i = 1; i < n; i++)

{

int get = A[i]; // 右手抓到一张扑克牌

int left = 0; // 拿在左手上的牌总是排序好的，所以可以用二分法

int right = i - 1; // 手牌左右边界进行初始化

while (left <= right) // 采用二分法定位新牌的位置

{

int mid = (left + right) / 2;

if (A[mid] > get)

right = mid - 1;

else

left = mid + 1;

}

for (int j = i - 1; j >= left; j--) // 将欲插入新牌位置右边的牌整体向右移动一个单位

{

A[j + 1] = A[j];

}

A[left] = get; // 将抓到的牌插入手牌

}

}

int main()

{

int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大二分插入排序

int n = sizeof(A) / sizeof(int);

InsertionSortDichotomy(A, n);

printf("二分插入排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

printf("\n");

return 0;

}

　　当n较大时，二分插入排序的比较次数比直接插入排序的最差情况好得多，但比直接插入排序的最好情况要差，所**当以元素初始序列已经接近升序时，直接插入排序比二分插入排序比较次数少**。二分插入排序元素**移动次数与直接插入排序相同**，依赖于元素初始序列。

### 插入排序的更高效改进：希尔排序(Shell Sort)

　　希尔排序，也叫**递减增量排序**，是插入排序的一种更高效的改进版本。希尔排序是**不稳定**的排序算法。

　　希尔排序是基于插入排序的以下两点性质而提出改进方法的：

插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率

但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位

　　希尔排序通过将比较的全部元素分为几个区域来提升插入排序的性能。这样可以让一个元素可以一次性地朝最终位置前进一大步。然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序，但是到了这步，需排序的数据几乎是已排好的了（此时插入排序较快）。  
　　假设有一个很小的数据在一个已按升序排好序的数组的末端。如果用复杂度为O(n^2)的排序（冒泡排序或直接插入排序），可能会进行n次的比较和交换才能将该数据移至正确位置。而希尔排序会用较大的步长移动数据，所以小数据只需进行少数比较和交换即可到正确位置。

#include <stdio.h>

// 分类 -------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- 根据步长序列的不同而不同。已知最好的为O(n(logn)^2)

// 最优时间复杂度 ---- O(n)

// 平均时间复杂度 ---- 根据步长序列的不同而不同。

// 所需辅助空间 ------ O(1)

// 稳定性 ------------ 不稳定

void ShellSort(int A[], int n)

{

int h = 0;

while (h <= n) // 生成初始增量

{

h = 3 \* h + 1;

}

while (h >= 1)

{

for (int i = h; i < n; i++)

{

int j = i - h;

int get = A[i];

while (j >= 0 && A[j] > get)

{

A[j + h] = A[j];

j = j - h;

}

A[j + h] = get;

}

h = (h - 1) / 3; // 递减增量

}

}

int main()

{

int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大希尔排序

int n = sizeof(A) / sizeof(int);

ShellSort(A, n);

printf("希尔排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

printf("\n");

return 0;

}

**希尔排序是不稳定的排序算法，**虽然一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱。

比如序列：{ 3, 5, 10, 8, 7, 2, 8, 1, 20, 6 }，h=2时分成两个子序列 { 3, 10, 7, 8, 20 } 和  { 5, 8, 2, 1, 6 } ，未排序之前第二个子序列中的8在前面，现在对两个子序列进行插入排序，得到 { 3, 7, 8, 10, 20 } 和 { 1, 2, 5, 6, 8 } ，即 { 3, 1, 7, 2, 8, 5, 10, 6, 20, 8 } ，两个8的相对次序发生了改变。

### 归并排序(Merge Sort)

　　归并排序是创建在归并操作上的一种有效的排序算法，效率为O(nlogn)，1945年由冯·诺伊曼首次提出。

　　归并排序的实现分为**递归实现**与**非递归(迭代)实现**。**递归实现**的归并排序是算法设计中**分治策略**的典型应用，我们将一个大问题分割成小问题分别解决，然后用**所有小问题的答案来解决整个大问题**。**非递归(迭代)实现的归并排序**首先进行是两两归并，然后四四归并，然后是八八归并，一直下去直到归并了整个数组。

　　归并排序算法主要依赖归并(Merge)操作。归并操作指的是将两个已经排序的序列合并成一个序列的操作，**归并操作**步骤如下：

**申请空间**，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列

设定两个指针，最初位置分别为两个已经排序序列的起始位置

比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置

重复步骤3直到某一指针到达序列尾

将另一序列剩下的所有元素直接复制到合并序列尾

　　归并排序的代码如下：

#include <stdio.h>

#include <limits.h>

// 分类 -------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- O(nlogn)

// 最优时间复杂度 ---- O(nlogn)

// 平均时间复杂度 ---- O(nlogn)

// 所需辅助空间 ------ O(n)

// 稳定性 ------------ 稳定

void Merge(int A[], int left, int mid, int right)// 合并两个已排好序的数组A[left...mid]和A[mid+1...right]

{

int len = right - left + 1;

int \*temp = new int[len]; // 辅助空间O(n)

int index = 0;

int i = left; // 前一数组的起始元素

int j = mid + 1; // 后一数组的起始元素

while (i <= mid && j <= right)

{

temp[index++] = A[i] <= A[j] ? A[i++] : A[j++]; // 带等号保证归并排序的稳定性

}

while (i <= mid)

{

temp[index++] = A[i++];

}

while (j <= right)

{

temp[index++] = A[j++];

}

for (int k = 0; k < len; k++)

{

A[left++] = temp[k];

}

}

void MergeSortRecursion(int A[], int left, int right) // 递归实现的归并排序(自顶向下)

{

if (left == right) // 当待排序的序列长度为1时，递归开始回溯，进行merge操作

return;

int mid = (left + right) / 2;

MergeSortRecursion(A, left, mid);

MergeSortRecursion(A, mid + 1, right);

Merge(A, left, mid, right);

}

void MergeSortIteration(int A[], int len) // 非递归(迭代)实现的归并排序(自底向上)

{

int left, mid, right;// 子数组索引,前一个为A[left...mid]，后一个子数组为A[mid+1...right]

for (int i = 1; i < len; i \*= 2) // 子数组的大小i初始为1，每轮翻倍

{

left = 0;

while (left + i < len) // 后一个子数组存在(需要归并)

{

mid = left + i - 1;

right = mid + i < len ? mid + i : len - 1;// 后一个子数组大小可能不够

Merge(A, left, mid, right);

left = right + 1; // 前一个子数组索引向后移动

}

}

}

int main()

{

int A1[] = { 6, 5, 3, 1, 8, 7, 2, 4 }; // 从小到大归并排序

int A2[] = { 6, 5, 3, 1, 8, 7, 2, 4 };

int n1 = sizeof(A1) / sizeof(int);

int n2 = sizeof(A2) / sizeof(int);

MergeSortRecursion(A1, 0, n1 - 1); // 递归实现

MergeSortIteration(A2, n2); // 非递归实现

printf("递归实现的归并排序结果：");

for (int i = 0; i < n1; i++)

{

printf("%d ", A1[i]);

}

printf("\n");

printf("非递归实现的归并排序结果：");

for (int i = 0; i < n2; i++)

{

printf("%d ", A2[i]);

}

printf("\n");

return 0;

}

　　上述代码对序列{ 6, 5, 3, 1, 8, 7, 2, 4 }进行归并排序的实例

　　归并排序除了可以对数组进行排序，还可以高效的求出数组小和（即单调和）以及数组中的逆序对。

### 堆排序(Heap Sort)

　 堆排序是指利用堆这种数据结构所设计的一种选择排序算法。堆是一种近似完全二叉树的结构（通常堆是通过一维数组来实现的），并满足性质：以最大堆（也叫大根堆、大顶堆）为例，其中父结点的值总是大于它的孩子节点。

　　我们可以很容易的定义堆排序的过程：

由输入的无序数组构造一个最大堆，作为初始的无序区

把堆顶元素（最大值）和堆尾元素互换

把堆（无序区）的尺寸缩小1，并调用heapify(A, 0)从新的堆顶元素开始进行堆调整

重复步骤2，直到堆的尺寸为1

　　堆排序的代码如下：

#include <stdio.h>

// 分类 -------------- 内部比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- O(nlogn)

// 最优时间复杂度 ---- O(nlogn)

// 平均时间复杂度 ---- O(nlogn)

// 所需辅助空间 ------ O(1)

// 稳定性 ------------ 不稳定

void Swap(int A[], int i, int j)

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

void Heapify(int A[], int i, int size) // 从A[i]向下进行堆调整

{

int left\_child = 2 \* i + 1; // 左孩子索引

int right\_child = 2 \* i + 2; // 右孩子索引

int max = i; // 选出当前结点与其左右孩子三者之中的最大值

if (left\_child < size && A[left\_child] > A[max])

max = left\_child;

if (right\_child < size && A[right\_child] > A[max])

max = right\_child;

if (max != i)

{

Swap(A, i, max); // 把当前结点和它的最大(直接)子节点进行交换

Heapify(A, max, size); // 递归调用，继续从当前结点向下进行堆调整

}

}

int BuildHeap(int A[], int n) // 建堆，时间复杂度O(n)

{

int heap\_size = n;

for (int i = heap\_size / 2 - 1; i >= 0; i--) // 从每一个非叶结点开始向下进行堆调整

Heapify(A, i, heap\_size);

return heap\_size;

}

void HeapSort(int A[], int n)

{

int heap\_size = BuildHeap(A, n); // 建立一个最大堆

while (heap\_size > 1) 　　　　　// 堆（无序区）元素个数大于1，未完成排序

{

// 将堆顶元素与堆的最后一个元素互换，并从堆中去掉最后一个元素

// 此处交换操作很有可能把后面元素的稳定性打乱，所以**堆排序是不稳定的排序算法**

Swap(A, 0, --heap\_size);

Heapify(A, 0, heap\_size); // 从新的堆顶元素开始向下进行堆调整，时间复杂度O(logn)

}

}

int main()

{

int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大堆排序

int n = sizeof(A) / sizeof(int);

HeapSort(A, n);

printf("堆排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

printf("\n");

return 0;

}

**堆排序是不稳定的排序算法，不稳定发生在堆顶元素与A[i]交换的时刻。**

比如序列：{ 9, 5, 7, 5 }，堆顶元素是9，堆排序下一步将9和第二个5进行交换，得到序列 { 5, 5, 7, 9 }，再进行堆调整得到{ 7, 5, 5, 9 }，重复之前的操作最后得到{ 5, 5, 7, 9 }从而改变了两个5的相对次序。

堆排序图解详见：https://www.cnblogs.com/MOBIN/p/5374217.html

### 快速排序(Quick Sort)

　　快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序n个元素要O(nlogn)次比较。在最坏状况下则需要O(n^2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他O(nlogn)算法更快，因为它的内部循环可以在大部分的架构上很有效率地被实现出来。

**快速排序使用分治策略**(Divide and Conquer)来把一个序列分为两个子序列。步骤为：

1. 从序列中挑出一个元素，作为"基准"(pivot).
2. 把所有比基准值小的元素放在基准前面，所有比基准值大的元素放在基准的后面（相同的数可以到任一边），这个称为分区(partition)操作。
3. 对每个分区递归地进行步骤1~2，递归的结束条件是序列的大小是0或1，这时整体已经被排好序了。

　　快速排序的代码如下：

#include <stdio.h>

// 分类 ------------ 内部比较排序

// 数据结构 --------- 数组

// 最差时间复杂度 ---- 每次选取的基准都是最大（或最小）的元素，导致每次只划分出了一个分区，需要进行n-1次划分才能结束递归，时间复杂度为O(n^2)

// 最优时间复杂度 ---- 每次选取的基准都是中位数，这样每次都均匀的划分出两个分区，只需要logn次划分就能结束递归，时间复杂度为O(nlogn)

// 平均时间复杂度 ---- O(nlogn)

// 所需辅助空间 ------ 主要是递归造成的栈空间的使用(用来保存left和right等局部变量)，取决于递归树的深度，一般为O(logn)，最差为O(n)

// 稳定性 ---------- 不稳定

void Swap(int A[], int i, int j)

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

int Partition(int A[], int left, int right) // 划分函数

{

int pivot = A[right]; // 这里每次都选择最后一个元素作为基准

int tail = left - 1; // tail为小于基准的子数组最后一个元素的索引

for (int i = left; i < right; i++) // 遍历基准以外的其他元素

{

if (A[i] <= pivot) // 把小于等于基准的元素放到前一个子数组末尾

{

Swap(A, ++tail, i);

}

}

Swap(A, tail + 1, right); // 最后把基准放到前一个子数组的后边，剩下的子数组既是大于基准的子数组

// 该操作很有可能把后面元素的稳定性打乱，所以快速排序是不稳定的排序算法

return tail + 1; // 返回基准的索引

}

void QuickSort(int A[], int left, int right)

{

if (left >= right)

return;

int pivot\_index = Partition(A, left, right); // 基准的索引

QuickSort(A, left, pivot\_index - 1);

QuickSort(A, pivot\_index + 1, right);

}

int main()

{

int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 }; // 从小到大快速排序

int n = sizeof(A) / sizeof(int);

QuickSort(A, 0, n - 1);

printf("快速排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

printf("\n");

return 0;

}

**快速排序是不稳定的排序算法，不稳定发生在基准元素与A[tail+1]交换的时刻。**

比如序列：{ 1, 3, 4, 2, 8, 9, 8, 7, 5 }，基准元素是5，一次划分操作后5要和第一个8进行交换，从而改变了两个元素8的相对次序。

　　Java系统提供的Arrays.sort函数。对于基础类型，底层使用快速排序。对于非基础类型，底层使用归并排序。请问是为什么？

　　答：这是考虑到排序算法的稳定性。对于基础类型，相同值是无差别的，排序前后相同值的相对位置并不重要，所以选择更为高效的快速排序，尽管它是不稳定的排序算法；而对于非基础类型，排序前后相等实例的相对位置不宜改变，所以选择稳定的归并排序。

### 非比较排序目录

* [计数排序](http://www.cnblogs.com/eniac12/p/5332117.html#s1)
* [基数排序](http://www.cnblogs.com/eniac12/p/5332117.html#s2)
* [桶排序](http://www.cnblogs.com/eniac12/p/5332117.html#s3)

[上一篇文章](http://www.cnblogs.com/eniac12/p/5329396.html)中我们总结了常用的比较排序算法，主要有**冒泡排序**，**选择排序**，**插入排序**，**归并排序**，**堆排序**，**快速排序**等。

　　这篇文章中我们来探讨一下常用的非比较排序算法：**计数排序**，**基数排序**，**桶排序**。在一定条件下，它们的时间复杂度可以达到O(n)。

　　这里我们用到的唯一数据结构就是数组，当然我们也可以利用链表来实现下述算法。

### 计数排序(Counting Sort)

　　计数排序用到一个额外的计数数组C，根据数组C来将原数组A中的元素排到正确的位置。

　　通俗地理解，例如有10个年龄不同的人，假如统计出有8个人的年龄不比小明大（即小于等于小明的年龄，这里也包括了小明），那么小明的年龄就排在第8位，通过这种思想可以确定每个人的位置，也就排好了序。当然，年龄一样时需要特殊处理（保证稳定性）：通过反向填充目标数组，填充完毕后将对应的数字统计递减，可以确保计数排序的稳定性。

　　计数排序的步骤如下：

1. 统计数组A中每个值A[i]出现的次数，存入C[A[i]]
2. 从前向后，使数组C中的每个值等于其与前一项相加，这样数组C[A[i]]就变成了代表数组A中小于等于A[i]的元素个数
3. 反向填充目标数组B：将数组元素A[i]放在数组B的第C[A[i]]个位置（下标为C[A[i]] - 1），每放一个元素就将C[A[i]]递减

　　计数排序的实现代码如下：

#include<iostream>

using namespace std;

// 分类 ------------ 内部非比较排序

// 数据结构 --------- 数组

// 最差时间复杂度 ---- O(n + k)

// 最优时间复杂度 ---- O(n + k)

// 平均时间复杂度 ---- O(n + k)

// 所需辅助空间 ------ O(n + k)

// 稳定性 ----------- 稳定

const int k = 100; // 基数为100，排序[0,99]内的整数

int C[k]; // 计数数组

void CountingSort(int A[], int n)

{

for (int i = 0; i < k; i++) // 初始化，将数组C中的元素置0(此步骤可省略，整型数组元素默认值为0)

{

C[i] = 0;

}

for (int i = 0; i < n; i++) // 使C[i]保存着等于i的元素个数

{

C[A[i]]++;

}

for (int i = 1; i < k; i++) // 使C[i]保存着小于等于i的元素个数，排序后元素i就放在第C[i]个输出位置上

{

C[i] = C[i] + C[i - 1];

}

int \*B = (int \*)malloc((n) \* sizeof(int));// 分配临时空间,长度为n，用来暂存中间数据

for (int i = n - 1; i >= 0; i--) // 从后向前扫描保证计数排序的稳定性(重复元素相对次序不变)

{

B[--C[A[i]]] = A[i]; // 把每个元素A[i]放到它在输出数组B中的正确位置上

// 当再遇到重复元素时会被放在当前元素的前一个位置上保证计数排序的稳定性

}

for (int i = 0; i < n; i++) // 把临时空间B中的数据拷贝回A

{

A[i] = B[i];

}

free(B); // 释放临时空间

}

int main()

{

int A[] = { 15, 22, 19, 46, 27, 73, 1, 19, 8 }; // 针对计数排序设计的输入，每一个元素都在[0,100]上且有重复元素

int n = sizeof(A) / sizeof(int);

CountingSort(A, n);

printf("计数排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

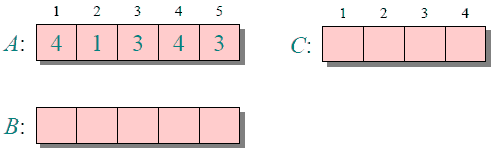
}

printf("\n");

return 0;

}

　　下图给出了对{ 4, 1, 3, 4, 3 }进行计数排序的简单演示过程



　　计数排序的时间复杂度和空间复杂度与**数组A的数据范围**（A中元素的最大值与最小值的差加上1）有关，因此**对于数据范围很大的数组，计数排序需要大量时间和内存。**

　　例如：对0到99之间的数字进行排序，计数排序是最好的算法，然而计数排序并不适合按字母顺序排序人名，**将计数排序用在基数排序算法中，能够更有效的排序数据范围很大的数组。**

### 基数排序(Radix Sort)

　　基数排序的发明可以追溯到1887年赫尔曼·何乐礼在打孔卡片制表机上的贡献。它是这样实现的：将所有待比较正整数统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始进行基数为10的计数排序，一直到最高位计数排序完后，数列就变成一个有序序列（利用了计数排序的稳定性）。

　　基数排序的实现代码如下：

#include<iostream>

using namespace std;

// 分类 ------------- 内部非比较排序

// 数据结构 ---------- 数组

// 最差时间复杂度 ---- O(n \* dn)

// 最优时间复杂度 ---- O(n \* dn)

// 平均时间复杂度 ---- O(n \* dn)

// 所需辅助空间 ------ O(n \* dn)

// 稳定性 ----------- 稳定

const int dn = 3; // 待排序的元素为三位数及以下

const int k = 10; // 基数为10，每一位的数字都是[0,9]内的整数

int C[k];

int GetDigit(int x, int d) // 获得元素x的第d位数字

{

int radix[] = { 1, 1, 10, 100 };// 最大为三位数，所以这里只要到百位就满足了

return (x / radix[d]) % 10;

}

void CountingSort(int A[], int n, int d)// 依据元素的第d位数字，对A数组进行计数排序

{

for (int i = 0; i < k; i++)

{

C[i] = 0;

}

for (int i = 0; i < n; i++) // n个数各自的第d位数字+1 为下标的C[ ]数值+1

{

C[GetDigit(A[i], d)]++;

}

for (int i = 1; i < k; i++)

{

C[i] = C[i] + C[i - 1]; //出现次数转换为排序位置

}

int \*B = (int\*)malloc(n \* sizeof(int));

for (int i = n - 1; i >= 0; i--)

{

int dight = GetDigit(A[i], d); // 元素A[i]当前位数字为dight

B[--C[dight]] = A[i]; // 根据当前位数字，把每个元素A[i]放到它在输出数组B中的正确位置上

// 当再遇到当前位数字同为dight的元素时，会将其放在当前元素的前一个位置上保证计数排序的稳定性

}

for (int i = 0; i < n; i++)

{

A[i] = B[i];

}

free(B);

}

void LsdRadixSort(int A[], int n) // 最低位优先基数排序

{

for (int d = 1; d <= dn; d++) // 从低位到高位

CountingSort(A, n, d); // 依据第d位数字对A进行计数排序

}

int main()

{

int A[] = { 20, 90, 64, 289, 998, 365, 852, 123, 789, 456 };// 针对基数排序设计的输入

int n = sizeof(A) / sizeof(int);

LsdRadixSort(A, n);

printf("基数排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

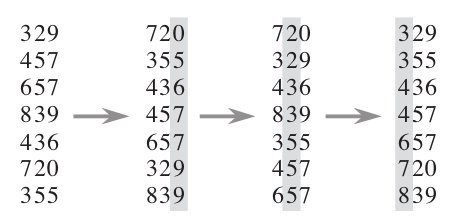
}

printf("\n");

return 0;

}

　　下图给出了对{ 329, 457, 657, 839, 436, 720, 355 }进行基数排序的简单演示过程



　　基数排序的时间复杂度是O(n \* dn)，其中n是待排序元素个数，dn是数字位数。这个时间复杂度不一定优于O(n log n)，dn的大小取决于数字位的选择（比如比特位数），和待排序数据所属数据类型的全集的大小；dn决定了进行多少轮处理，而n是每轮处理的操作数目。

　　如果考虑和比较排序进行对照，基数排序的形式复杂度虽然不一定更小，但由于不进行比较，因此其基本操作的代价较小，而且如果适当的选择基数，dn一般不大于log n，所以**基数排序一般要快过基于比较的排序，比如快速排序**。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以**基数排序并不是只能用于整数排序**。

### 桶排序(Bucket Sort)

 　　桶排序也叫箱排序。工作的原理是将数组元素映射到有限数量个桶里，利用计数排序可以定位桶的边界，每个桶再各自进行桶内排序（使用其它排序算法或以递归方式继续使用桶排序）。

　　桶排序的实现代码如下：

#include<iostream>

using namespace std;

// 分类 ------------- 内部非比较排序

// 数据结构 --------- 数组

// 最差时间复杂度 ---- O(nlogn)或O(n^2)，只有一个桶，取决于桶内排序方式

// 最优时间复杂度 ---- O(n)，每个元素占一个桶

// 平均时间复杂度 ---- O(n)，保证各个桶内元素个数均匀即可

// 所需辅助空间 ------ O(n + bn)

// 稳定性 ----------- 稳定

/\* 本程序用数组模拟桶 \*/

const int bn = 5; // 这里排序[0,49]的元素，使用5个桶就够了，也可以根据输入动态确定桶的数量

int C[bn]; // 计数数组，存放桶的边界信息

void InsertionSort(int A[], int left, int right)

{

for (int i = left + 1; i <= right; i++) // 从第二张牌开始抓，直到最后一张牌

{

int get = A[i];

int j = i - 1;

while (j >= left && A[j] > get)

{

A[j + 1] = A[j];

j--;

}

A[j + 1] = get;

}

}

int MapToBucket(int x)

{

return x / 10; // 映射函数f(x)，作用相当于快排中的Partition，把大量数据分割成基本有序的数据块

}

void CountingSort(int A[], int n)

{

for (int i = 0; i < bn; i++)

{

C[i] = 0;

}

for (int i = 0; i < n; i++) // 使C[i]保存着i号桶中元素的个数

{

C[MapToBucket(A[i])]++;

}

for (int i = 1; i < bn; i++) // 定位桶边界：初始时，C[i]-1为i号桶最后一个元素的位置

{

C[i] = C[i] + C[i - 1];

}

int \*B = (int \*)malloc((n) \* sizeof(int));

for (int i = n - 1; i >= 0; i--)// 从后向前扫描保证计数排序的稳定性(重复元素相对次序不变)

{

int b = MapToBucket(A[i]); // 元素A[i]位于b号桶

B[--C[b]] = A[i]; // 把每个元素A[i]放到它在输出数组B中的正确位置上

// 桶的边界被更新：C[b]为b号桶第一个元素的位置

}

for (int i = 0; i < n; i++)

{

A[i] = B[i];

}

free(B);

}

void BucketSort(int A[], int n)

{

CountingSort(A, n); // 利用计数排序确定各个桶的边界（分桶）

for (int i = 0; i < bn; i++) // 对每一个桶中的元素应用插入排序

{

int left = C[i]; // C[i]为i号桶第一个元素的位置

int right = (i == bn - 1 ? n - 1 : C[i + 1] - 1);// C[i+1]-1为i号桶最后一个元素的位置

if (left < right) // 对元素个数大于1的桶进行桶内插入排序

InsertionSort(A, left, right);

}

}

int main()

{

int A[] = { 29, 25, 3, 49, 9, 37, 21, 43 };// 针对桶排序设计的输入

int n = sizeof(A) / sizeof(int);

BucketSort(A, n);

printf("桶排序结果：");

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

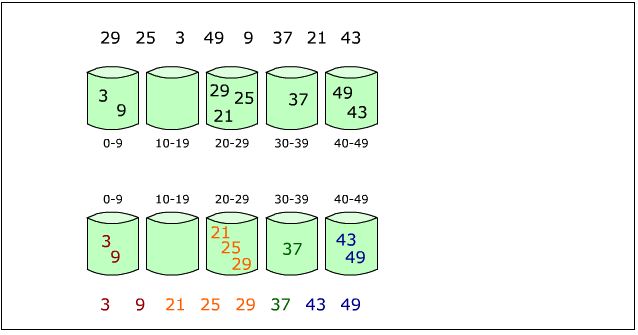
printf("\n");

return 0;

}

[复制代码](javascript:void(0);)

下图给出了对{ 29, 25, 3, 49, 9, 37, 21, 43 }进行桶排序的简单演示过程



　　桶排序不是比较排序，不受到O(nlogn)下限的影响，它是鸽巢排序的一种归纳结果，当所要排序的数组值分散均匀的时候，桶排序拥有线性的时间复杂度。

链接指示：extern “C”(作用)

C++程序有时需要**调用其他语言编写的函数**，最常见的是调用C语言编写的函数。像所有其他名字一样，**其他语言中的函数名字也必须在C++中进行声明**，并且**该声明必须指定返回类型和形参列表**。对于其他语言编写的函数来说，编译器检查其调用方式与处理普通C++函数的方式相同，但生成的代码有所区别。C++使用链接指示（linkage directive）指出任意非C++函数所用的语言。

　　声明一个非C++函数：

**链接指示可以有两种形式：单个或复合**。链接指示**不能出现在类定义或函数定义的内部**。

　　例如：

　　单语句：　　extern "C" size\_t strlen(const char \*);

　　复合语句：extern "C" {

　　　　　　int strcmp(const char\*, const char\*);

　　　　　　char \*strcat(char\*, const char\*);

　　}

　　链接指示与头文件：  
　　复合语句：

　　extern "C" {

　　　　#include <string.h>

　　}

**指向extern "C"函数的指针**：

　　编写函数所用的语言是函数类型的一部分。（指向其他语言编写的函数的指针必须与函数本身使用相同的链接指示）

**extern "C" void (\*pf)(int)；**

　　当我们使用pf调用函数时，编译器认定当前调用的是一个C函数。

　　指向C函数的指针与指向C++函数的指针是不一样的类型。

　　链接指示对整个声明都有效：

　　当我们使用链接指示时，他不仅对函数有效，而且对作为返回类型或形参类型的函数指针也有效。

　　//f1是一个C函数，它的形参是一个指向C函数的指针

　　extern "C" void f1( void(\*)(int) );

　　因为链接指示同时作用于声明语句中的所有函数，所以如果我们希望给C++函数传入一个指向C函数的指针，则必须使用类型别名。

　　//FC是一个指向C函数的指针

　　extern "C" typedef void FC( int );

　　//f2是一个C++函数，该函数的形参是指向C函数的指针

　　void f2(FC \*);

　　导出C++函数到其他语言：

　　通过使用链接指示对函数进行定义，我们可以令一个C++函数在其他语言编写的程序中可用。

　　//calc函数可以被C程序调用

　　extern "C" double calc( double dparm ) {/\*......\*/}

　　编译器将为该函数生成适合指定语言的代码

　　对链接到C的预处理器的支持

　　有时需要在C和C++中编译同一个源文件，为了实现这一目的，在编译C++版本的程序时预处理器定义 \_\_cplusplus(两个下划线)。利用这个变量，我们可以在编译C++程序的时候有条件地包含进来一些代码：

　　#ifndef \_\_cplusplus

　　//正确：我们在编译C++程序

　　extern "C"

　　#endif

　　int strcmp( const char\*, const char\* );

　　重载函数与链接指示：

**C语言不支持函数重载，因此也不难理解为什么一个C链接指示只能用于说明一组重载函数中的某一行**了：

　　//错误：两个extern "C"函数的名字相同

　　extern "C" void print ( const char\* );

　　extern "C" void print ( int );

　　如果在一组重载函数中有一个C函数，则其余函数必定都是C++函数。

C语言和C++有什么区别？（大体讲一下，继承、多态、封装、异常处理等）

差不多是win98跟winXP的关系。C++是在C的基础上增加了新的理论，玩出了新的花样。所以叫C加加。  
**C是一个结构化语言，它的重点在于算法和数据结构**。C程序的设计首要考虑的是如何通过一个过程，对输入（或环境条件）进行运算处理得到输出（或实现过程（事务）控制）。  
**C++，首要考虑的是如何构造一个对象模型**，让这个模型能够契合与之对应的问题域，这样就可以通过获取对象的状态信息得到输出或实现过程（事务）控制。 所以C与C++的最大区别在于它们的用于**解决问题的思想方法不一样**。之所以说C++比C更先进，是因为“ 设计这个概念已经被融入到C++之中 ”。

C语言与C＋＋的区别有很多，下面是简要概述：

1、全新的程序程序思维，C语言是面向过程的，而C＋＋是面向对象的。  
  
2、**C语言有标准的函数库，它们松散的**，只是把功能相同的函数放在一个头文件中；而**C++对于大多数的函数都是有集成的很紧密**，特别是C语言中没有的C++中的API是对Window系统的大多数API有机的组合，是一个集体。但你也可能单独调用API。  
  
3、特别是C++中的图形处理，它和C语言的图形有很大的区别。C语言中的图形处理函数基本上是不能用在C++中的。**C语言标准中不包括图形处理**。  
  
4、**C和C++中都有结构的概念，但是在C语言中结构只有成员变量，而没成员方法，而在C++中结构中，它可以有自己的成员变量和成员函数**。但是在C语言中结构的成员是公共的，什么想访问它的都可以访问；而在VC++中它没有加限定符的为私有的。  
  
5、C语言可以写很多方面的程序，但是**C++可以写得更多更好**，C++可以写基于DOS的程序，写DLL，写控件，写系统。  
  
6、C语言对程序的文件的组织是松散的，几乎是全要程序处理；而**c++对文件的组织是以工程，各文件分类明确**。  
  
7、**C++中的IDE很智能**，和VB一样，有的功能可能比VB还强。  
  
8、**C++对可以自动生成你想要的程序结构使你可以省很多时间**。有很多可用工具如加入MFC中的类的时候，加入变量的时候等等。  
  
9、**C++中的附加工具也有很多**，可以进行系统的分析，可以查看API；可以查看控件。  
  
10、**调试功能强大，并且方法多样**。

首先这个问题问的就没太大意义。因为 C++ 把 C 作为它的子集。那么作为一个问题，还是要大概回答一下的。换句话说，C++ 在 C 的基础上引入了那些新的东西？

简而言之，就是 C++ 比 C 多了 class，所以 C++ 里引入了面向对象思想，产生了面向接口编程，比如说微软的 COM 技术。C++ 引入模板，将对象类型抽象化和独立，产生了通用编程思想。这是两种语言的区别所在。

由于有了面向对象思想，所以在语言层面，C 中的数据，方法都是平面的，开放的，零散的，缺乏组织层次关系的，但也是非常明确的，静态的。而 C++ 可以做到数据和方法的封装，使得 C++ 在语言层面可以有组织，有层次，关系也是隐晦的。同时，C++ 引入对象后也引入了更多复杂性，编译器需要做的事情更多，更复杂。

C++ 作者原想让 C++ 成为一个独立的新语言，例如，C++ 完全不想要 struct 。但是 C 语言的遗产太多且根深蒂固，所以 C++ 将 C 语言已有基础都继承了下来。

C++有以下几个范型：  
1，过程形式 2，面向对象形式 3，函数形式 4，泛型形式 5，元编程形式。你看，这就超过C的范畴了。

可以认为C++有四个次语言：  
C. 2，面向对象C++ 3，模板C++ 4， STL

以下回答印象分情况:  
c面向过程，c++面向对象(扣分)  
c适合开发比c++还底层的东西，如操作系统，协议栈。(对百度来说不扣不减)  
精通c++的，跟c完全是2种语言，除了基本类型，条件循环语句，等差不多外，在编程思维上完全2种东西。c++思考的是对象，c思考的是寄存器，堆栈。没有很好掌握的，c++就是比c还容易出错的c，简称c++(加分)  
c能做的，c++也能做，在一定程度上，c++包含c(扣分)

网易二面：

1. qt类中的一些继承关系？
2. Qt的信号与槽机制？
3. Qt有哪些类？控件？
4. PLC是个什么玩意？
5. Strcpy函数的编写？（阿里校招也让现场编写了）
6. 数据结构中二叉树的非递归遍历？（画图举例讲解）
7. C++中四种类型转换机制？
8. 继承机制中对象之间是如何转换的？
9. 继承机制中引用和指针之间如何转换？
10. 虚函数，虚函数表里面内存如何分配？
11. 如何实现只能动态/静态分配对象，不能定义类对象？
12. stl有哪些容器，对比vector和set？
13. 红黑树的定义和解释？
14. Const关键字的作用？（const成员函数，函数传递，和define的区别）
15. 静态成员函数和数据成员有什么意义？
16. 模板特化的概念，为什么特化？
17. Explicit是干什么用的？
18. Strcpy返回类型是干嘛用的？
19. 内存溢出有哪些因素？
20. New与malloc的区别，delete和free的区别？
21. 为什么要用static\_cast转换而不用C语言中的转换？
22. 异常机制是怎么回事？
23. 迭代器删除元素会发生什么？
24. 必须再构造函数初始化式里进行初始化的数据成员有哪些？
25. 类的封装：private，protected。Public
26. Auto\_ptr类？