C++程序设计精要教程

华中科技大学

第12章 类型解析、转换与推导

◆12.1 隐式与显式类型转换

- **X86编译模式简单类型字节数：sizeof(bool)≤sizeof(char)≤sizeof(short)≤sizeof(int)≤sizeof (long)≤sizeof(float)≤sizeof(double)≤sizeof(long double) 。**
- **字节数少的类型向字节数多的类型转换时，一般不会引起数据的精度损失。**
- **无风险的转换由编译程序自动完成，这种不提示程序员的自动转换也称为隐式类型转换。**
- **隐式转换的基本方式: (1)非浮点类型字节少的向字节数多的转换;(2)非浮点类型有符号数向无符号数转换;(3)运算时整数向double类型的转换。**
- **默认时，bool、char、short和int的运算按int类型进行，所有浮点常量及浮点数的运算按double类型进行。**
- **赋值或调用时参数传递的类型相容，是指可以隐式转换，包括父子类的相容。**

第12章 类型解析、转换与推导

【例12.2】实参传递给函数时应和形参类型相同或相容。

```
#include <iostream>
using namespace std;
double area(double r)
{
    return 3.14159*r*r;          //注意浮点常量3.14159，默认其为double类型
}
void main( )
{
    char m=6556806;              //6556790=0x640c86，截断后x=0x86=-122;
    int x=2;                     //常量2被编译程序默认当作int类型
    double a=area(x);            //形参r的类型和实参x的类型相容：可自动转换
    a=area('A');                 //字符'A'最终自动转换为double类型：类型相容
    cout<<"Area="<<a;            //常量"Area="的类型默认为const char*类型
}
```

第12章 类型解析、转换与推导

◆12.1 隐式与显式类型转换

- **可以设置VS2019给出最严格的编程检查：例如任何警告都报错等等。**
- **有关类型转换若有警告，则应修改为强制类型转换即显式类型转换。**
- **强制类型转换引起的问题由程序员自己负责。**

```
char u = 'a';                //编译时可计算，无截断，不报警
char v = 'a' + 1;            //编译时可计算，无截断，不报警
char w = 300;                //编译时可计算，有截断，要报警
int  x = 2;                  //x占用的字节数比char和short类型多，不报警
char y = x;                  //编译时不可计算，可能截断，要报错
short z = x;                 //编译时不可计算，可能截断，要报错
```

第12章 类型解析、转换与推导

◆12.1 隐式与显式类型转换

- **一般简单类型之间的强制类型转换的结果为右值。**
- **如果对可写变量进行同类型的左值引用转换，则转换结果为左值。**
- **只读的简单类型变量如果转换为可写左值，并不能修改其值(受到页面保护机制的保护）。**

```
int x=0;
((short) x)=2;      //报错：转换后((short) x)为传统右值，故不能出现在等号的左边
((int) x) =7;       //VS2019报错：传统右值不能出现在等号的左边。
((int &)x)=8;       //正确：x=8，用的不是最基本的简单类型，而是引用类型int &
const  int y=9;
((int &)y)=10;    //y的结果仍然为9，全局变量如此赋值可引起程序异常（内存页面保护）
```

第12章 类型解析、转换与推导

【例12.5】简单类型只读自动变量的强制类型转换与赋值。

```
#include <iostream>
using namespace std;
void main(int argc, char *argv[ ]) {
    const int x = 0;
    *(int *)&x = 2;                    //debug时x的值变为2
    cout << "x=" << x<< endl;          //但输出结果x=0不变
    int y = x;
    cout << "y=" << y<< endl;          //输出结果y=0
    const_cast<int &>(x) = 10;         //等价于*(int *)&x = 10
    y = x;
    cout << "y=" << y << endl;         //输出结果y=0
}
```

程序的输出结果如下。

```
x=0
y=0
y=0
```

第12章 类型解析、转换与推导

◆12.1 隐式与显式类型转换

- 但对于类的只读数据成员，如果转换为可写左值，可以修改其值。
- 目前操作系统并不支持分层保护机制，无法在对象层和数据成员层提供不同类型的保护。 【例12.4】

```
struct T {                  //例12.4
    int  x=0;               //有的成员可写
    const int y = 0;        //有的成员不可写
    int q( ) {
        *(int*)&y = y + 1;  return y;
    }
};
void main( ) {
    T m;  const T n;    //有的对象可写，有的对象不可写
    int x = m.q( );
    x = m.q( );
}
```

第12章 类型解析、转换与推导

◆12.2 cast系列类型转换

- **static_cast同C语言的强制类型转换用法基本相同，在转换目标为左值时不能从源类型中去除const和volitale属性，不做多态相关的检查。**
- **const_cast同C语言的强制类型转换用法基本相同，能从源类型中去除const和volitale属性。**
- **dynamic_cast将子类对象转换为父类对象时无须子类多态，而将基类对象转换为派生类对象时要求基类多态。**
- **reinterpret_cast主要用于名字同指针或引用类型之间的转换，以及指针与足够大的整数类型之间的转换。**

第12章 类型解析、转换与推导

◆static_cast——静态转换

- 使用格式为“static_cast<T> (expr)”，用于将数值表达式expr的源类型转换为T目标类型。
- 目标类型不能包含存储位置类修饰符，如static、extern、auto、register等。
- static_cast仅在编译时静态检查源类型能否转换为T类型，运行时不做动态类型检查。
- static_cast不能去除源类型的const或volatile。即不能将指向const或volatile实体的指针(或引用)转换为指向非const或volatile实体的指针(或引用)。

第12章 类型解析、转换与推导

【例12.6】使用static_cast对数值表达式进行强制类型转换。

```
#include <iostream>
using namespace std;
const int x=0;                  //x为const全局变量，即只读全局变量，内存分配在受保护的区域
volatile int y=0;               //y为可写易变全局变量
void main( ) {
    const int z=0;              //z为const自动变量，内存分配在受保护的区域
    int w = static_cast<int>(x);          //正确：x有const但被忽略，因为现在只读x的值，转换目标为右值
    //static_cast<int>(x) = 0;            //错误：转换结果为右值不能对其赋值
    //static_cast<int&>(x) = 0;           //错误：不能去除x的const只读属性，转换目标为左值
    //static_cast<int>(w) = 0;            //错误：转换结果为右值不能对其赋值
    static_cast<int&>(w) = 0;             //正确：转换为有址传统左值引用，可被赋值
```

第12章 类型解析、转换与推导

```
    //*static_cast<int*>(&x) = 0;          //错误：无法去除const将源类型const int *转换为int *。转换用作左值
    //static_cast<int&>(y) = 0;            //错误：无法去除全局变量y的volatile属性
    const_cast<int&>(y) = 4;               //正确：可以去除全局变量y的volatile属性，::y=4
    const_cast<int&>(x) = 3;               //正确：但运行时出现页面保护访问冲突
    *const_cast<int*>(&x) = 3;             //正确：但运行时出现页面保护访问冲突
    *(int*)&x = 3;                         //正确：但运行时出现页面保护访问冲突
    *const_cast<int*>(&z) = 3;             //正确：运行无异常，但并不能修改z的值
    cout << "z=" << z << endl;             //输出z的值，仍然为z=0
    *(int*)&z = 3;                         //正确：运行无异常，但并不能修改z的值
    cout << "z=" << z << endl;             //输出z的值，仍然为z=0
}
```

第12章 类型解析、转换与推导

◆const_cast——只读转换

- **const_cast的使用格式为 “const_cast<类型表达式>(数值表达式)”。**
- **类型表达式不能包含存储位置类修饰符，如static、extern、auto、register等。**
- **static_cast仅在编译时静态检查源类型能否转换为T类型，运行时不做动态类型检查。**
- **static_cast不能去除源类型的const或volatile。即不能将指向const或volatile实体的指针(或引用)转换为指向非const或volatile实体的指针(或引用)。**
- **不能用const_cast将无址常量、位段访问、无址返回值转换为有址引用**

第12章 类型解析、转换与推导

【例12.7】const_cast只能转换为指针、引用或指向对象成员的指针类型。

```
class Test {
    int number;                    //对于可修改对象，该数据成员可被修改
public:
    const int nn;                  //无论何种对象，该只读数据成员都不可被写
    void dec( )const;              //const说明对象*this不可写，故其成员都不可写
    Test(int m): nn(m) { number = m; }
};
void Test::dec( )const {           //this的类型为const Test*const，故Test对象不可修改
    //number--;                    //错误：当前对象不可写，故每个成员都不可修改
    const_cast<Test*>(this)->number--;//this去除const变为Test*const，对象可写
    //nn--;                        //错误：nn为const，只读不可写
    const_cast<int&>(nn)--; //nn去除const后可写
}
```

第12章 类型解析、转换与推导

```
void main( ) {
    Test a(7);                                  //a.number=7，a.nn=7
    a.dec( );                                   //a.number=6，a.nn=6
    const int xx = 0;                           //xx源类型为const int
    const static int& yy=0;                     //yy源类型为const int&
    volatile int zz = 0;                        //zz 源类型为volatile int
    int ww=*const_cast<int*>(&xx)=2; //但并不能改变受保护的xx，ww=2，xx=0
    ww = xx;                                    //ww=xx=0：xx受保护，xx的值并未被修改
    a.*const_cast<int Test::*>(&Test::nn)=3;//对象实例成员不受保护可修改：a.nn=3
    ww = a.nn;                                  //ww=3
    const_cast<volatile int&>(yy) = 4;  //添加volatile，引用yy无内存不受保护，yy=4
    ww = yy;                                    //ww=4
    const_cast<int&>(zz)=6;                     //去除“zz”源类型“volatile int”中的volatile：zz=6
    //const_cast<const int&>(zz)=6;    //错误：添加const后成为传统右值，不能赋值
    ww=const_cast<const int&>(zz);    //正确：添加const后成为传统右值，ww=6
    ww=*const_cast<const int*>(&zz);//正确：添加const后成为传统右值，ww=6
    const_cast<volatile int&>(ww)=5;  //正确：添加volatile：ww=5
}
```

第12章 类型解析、转换与推导

◆dynamic_cast——动态转换

- 关键字dynamic_cast主要用于子类向父类转换，以及有虚函数的基类向派生类转换。被转换的表达式必须涉及类类型。
- 使用格式为“dynamic_cast<T> (expr)”，要求类型T是类的引用、类的指针或者void*类型，而expr的源类型必须是类的对象（常量或变量：向引用类型转换）、父类或者子类的引用或指针。
- dynamic_cast转换时不能去除数值表达式expr源类型中的const和volitale属性。
- 有址引用和无址引用之间不能相互转换。
- 被转换的基类对象必须包含虚函数或纯虚函数才能转换为派生类对象。最好先用typeid检查确保基类对象实际上就是派生类对象。

第12章 类型解析、转换与推导

【例12.11】运行时不能使用dynamic_cast将有址引用转换为无址引用。

```
#include <iostream>
using namespace std;
struct B {
    int m;
    B(int x): m(x) { }
    virtual void f( ) { cout << 'B'; }        //若无虚函数，  “dynamic_cast<D*>(&b)” 向下转换出错
};
struct D : public B {                         //B是父类，D是子类
    int n;
    D(int x, int y): B(x), n(y) { }
    void f( ) { cout << 'D'; }                //函数f()自动成为虚函数
};
void main( ) {
    B a(3);
    B &b=a;
```

第12章 类型解析、转换与推导

```
D c(5,7);
D &d=c;
D *pc1 = static_cast<D*>(&a);       //语法正确但为不安全的自上向下转换
pc1->f( );                          //输出B
D *pc2 = static_cast<D*>(&b);       //语法正确但为不安全的自上向下转换
pc2->f( );                          //输出B
D *pc3 = dynamic_cast<D*>(&a);      //若a无虚函数f( )，则自上向下转换错误
pc3->f( );                          //运行异常：pc3为nullptr（a非子类对象）
D *pc4 = dynamic_cast<D*>(&b);      //若b无虚函数f( )，则自上向下转换错误
pc4->f( );                          //运行异常：pc4为空指针（b非子类对象）
B *pb1 = dynamic_cast<D*>(&c);      //语法正确且为安全的自下向上赋值
pb1->f( );                          //输出D：正确的多态行为
B *pb2 = dynamic_cast<D*>(&d);      //语法正确且为安全的自下向上赋值
pb2->f( );                          //输出D：正确的多态行为
D &ra1 = static_cast<D&>(a);        //语法正确但不为安全的自上向下转换
ra1.f( );                           //输出B
```

第12章 类型解析、转换与推导

```
    D &ra2 = static_cast<D&>(b);        //语法正确但不为安全的自上向下转换
    ra2.f( );                           //输出B：根据虚函数入口地址表首址
    B &rc1 = dynamic_cast<D&>(c);       //语法正确且为安全的自下向上赋值
    rc1.f( );                           //输出D：正确的多态行为
    B &rc2 = dynamic_cast<D&>(d);       //语法正确且为安全的自下向上赋值
    rc2.f( );                           //输出D：正确的多态行为
    B &&rc3 = static_cast<D&&>(c);      //语法正确且为安全的自下向上赋值
    rc3.f( );                           //输出D：正确的多态行为
    B &&rc4 = dynamic_cast<D&&>(c);//语法正确且为安全的自下向上赋值
    rc4.f( );                           //输出D：正确的多态行为
    B &&rc5 = static_cast<D&&>(d);      //语法正确且为安全的自下向上赋值
    rc5.f( );                           //输出D：正确的多态行为
    B &rc6 = dynamic_cast<D&>(rc5); //正确：自上向下转换，自下向上赋值
    rc6.f( );                           //输出D：正确的多态行为
}
```

第12章 类型解析、转换与推导

◆reinterpret_cast——重释转换

- **关键字reinterpret_cast名字到指针或引用类型的转换、有址引用与无址引用之间的相互转换、以及指针与足够大的整数类型之间的相互转换。**
- **使用格式为 “reinterpret_cast <T> (expr)”，用于将数值表达式expr的值转换成T类型的值。 T类型不能是实例数据成员指针。**
- **转换为足够大的整数类型是指能够存储一个地址或者指针的整数类型，X86和X64的指针大小不同，X86使用int类型即可。**
- **当T为使用&或&&定义的引用类型时，expr必须是一个有址表达式。**
- **有址引用和无址引用之间可以相互转换。**

第12章 类型解析、转换与推导

```
#include <iostream>          //例12.13
using namespace std;
struct B {
    int m;
    static int n;                    //静态成员有真正的单元地址
    B(int x): m(x) { }
    static void e( ) { cout << 'E'; }        //静态函数成员有真正入口地址
    virtual void f( ) { cout << 'F'; }
};
int B::n = 0;
void main( ) {
    B a(1);
    B &b = a;                        //b有址引用a，共享a的内存
```

第12章 类型解析、转换与推导

```
B *e = reinterpret_cast <B *> (&a);//&a为B*类型，无须转换，e=&a
e = reinterpret_cast <B *> (&b);                //&b即&a，无须转换，e=&a
int f = reinterpret_cast <int> (e);             //指针e转为整型，赋给f
B *g = reinterpret_cast <B *> (f);              //整数f转为B *，赋值给g=&a
B &h = reinterpret_cast <B&> (a); //名字a转引用，等价于B &h=a
h.m = 2;                                        //h共享a的内存，h.m=b.m=a.m=2
B &&i = reinterpret_cast <B&&> (b);             //有址引用b转无址引用，i共享b引用的a
i.m=3;                                          //i.m=h.m=b.m=a.m=3
int *j = reinterpret_cast <int *>(&B::n);//&B::n的类型为int *，无须转换，j=&B::n
int &k=reinterpret_cast <int &>(B::n);          //名字B::n转引用，等价于int &k=B::n
k=6;                                            //k=B::n=i.n=h.n=b.n=a.n=6;
void (*l)( )=reinterpret_cast<void(*)( )>(&B::e);//&B::e类型为void(*)( )，无须转换
l = reinterpret_cast <void(*)( )> (B::e);       //结果同上：静态函数成员名即函数地址
```

第12章 类型解析、转换与推导

```
 void(&m)( )=reinterpret_cast<void(&)( )>(B::e);//名字B::e转引用，void(&m)( )=B::e
   m( );                                                    //等价于调用B::e( )，输出E
   void (B::*n)( )=reinterpret_cast<void (B::*)( )>(&B::f);  //&B::f的类型无须转换
   (a.*n)( );                                               //等价于调用a.f( )，输出F
   int B::*o = reinterpret_cast <int B::*> (&B::m);//&B::m的类型为int B::*，无须转换
   f=a.*o;                                                  //f=a.m=h.m=3
   B &&p = reinterpret_cast <B&&>(h);          //有址引用转无址引用p，p.m=h.m=a.m=3
   p.m = 4;                                                 //p.m=h.m=b.m=a.m=4
   B &q = reinterpret_cast <B&> (p); //无址引用转有址引用：B&q=a，q.m=a.m=4
   q.m = 5;                                                 //q.m=p.m=h.m=b.m=a.m=5
}
```

第12章 类型解析、转换与推导

◆12.3 类型转换实例

- C++的父类指针（或引用）可以直接指向（或引用）子类对象，但是通过父类指针（或引用）只能调用父类定义的成员函数。
- 武断或盲目地向下转换，然后访问派生类或子类成员，会引起一系列安全问题: (1)成员访问越界(如父类无子类的成员); (2)函数不存在(如父类无子类函数) 。
- 关键字typeid可以获得对象的真实类型标识：有==、!=、before、raw_name、hash_code等函数。
- typeid使用格式：（1）typeid(类型表达式)；（2）typeid(数值表达式)。
- typeid的返回结果是“const type_info&”类型，在使用typeid之前可先“#include <typeinfo>”，在std名字空间。

第12章 类型解析、转换与推导

【例12.14】用类型检查typeid保证转换安全性。

```
#include <typeinfo>
#include <iostream>
using namespace std;
struct B {
    int m;
    B(int x): m(x) { }
    virtual void f( ) { cout << 'B'; }
};
struct D: public B {                                    //基类B和派生类D满足父子关系
    int n;
    D(int x, int y): B(x), n(y) { }
    void f( ) { cout << 'D' << endl; }
    void g( ) { cout << 'G' << endl; }
};
```

第12章 类型解析、转换与推导

```
int main(int argc, char *argv[ ]) {
    B a(3);                                          //定义父类对象a
    B &b = a;
    D c(5, 7);                                       //定义子类对象c
    D &d = c;
    B *pb = &a;                                      //定义父类指针pb指向父类对象a
    D *pc(nullptr);                                  //定义子类指针pc并设为空指针
    if (argc < 2) pb = &c;
    if (typeid(*pb) == typeid(D)) {                  //判断父类指针是否指向子类对象
        pc = (D*)pb;                                 //C语言的强制转换
        pc = static_cast<D*>(pb);                    //静态强制转换，安全，因为pb指向D类
        pc = dynamic_cast<D*>(pb);                   //动态强制转换：向下转换B须有虚函数
        pc = reinterpret_cast<D*>(pb);               //重释类型转换，安全，因为pb指向D类
        pc->g( );                                    //输出G，不转换pb无法调用g( )
    }
    cout << typeid(pc).name( )<<endl;                //输出struct D*
    cout << typeid(*pc).name( ) << endl;             //输出struct D
    cout << typeid(B).before(typeid(D))<<endl; //输出1即布尔值真：B是D的基类
}
```

第12章 类型解析、转换与推导

◆12.3 类型转换实例

- **保留字explicit只能用于定义构造函数或类型转换实例函数成员， explicit定义的实例函数成员必须显式调用。**

```
class COMPLEX {
    double r, v;
public:
    explicit COMPLEX(double r1=0, double v1 = 0)
    { r = r1; v = v1; }
    COMPLEX operator+(const COMPLEX &c)const
    {       return COMPLEX(r + c.r, v + c.v);     };
    explicit operator double() { return r; }
}m(2,3);
```

未用explicit定义前:
(1) double d=m等价于d=m.operator double( )
(2) m+2.0等价于m+COMPLEX(2.0, 0.0)

使用explicit定义后:
(1) 不能定义d=m;
(2) 不能用m+2.0相加。

第12章 类型解析、转换与推导

◆12.4 自动类型推导

- 保留字auto在C++中用于类型推导。
- 可用于推导变量、各种函数的返回值、以及类的有const定义的静态数据成员的类型。
- 使用auto推导时，被推导实体不能出现类型说明，但是可以出现存储可变特性const、voilatile和存储位置特性如static、register。

第12章 类型解析、转换与推导

```
#include <stdio.h>              //例12.18
inline auto a( ) { return; }    //推导函数a的返回类型为void
auto b = 'A';                   //正确：推导定义 “char b= 'A';”
auto c = 1 + printf("a");       //正确：推导定义 “int c=1 + printf("a"); ”
auto d = 3.2;                   //正确：推导定义 “double d= 3.2;”
auto e = "abcd";                //正确：推导定义 “const char *e = "abcd";”
static int f = 0;               //正确：使用static须明确说明变量类型int
static auto x = 3;              //正确：推导定义 “static int x = 3;”
class A {
    auto const static m=3;
    inline auto const volatile static m=x;  //使用inline时可使用任意表达式初始化
};
void  main() {
    auto b = 'A';               //正确：推导定义 “char b= 'A';”
    auto static x = 3;          //正确：推导定义 “static int x = 3;”
}
```

第12章 类型解析、转换与推导

◆12.4 自动类型推导

- 保留字auto可以推导与数组和函数相关的类型。
- 使用数组名代表整个数组类型，使用函数名代表该函数的指针。
- 使用“数组名[表达式]”表示除第一维外的数组类型，依此类推。
- 当被推导变量前面有“*”时，数组类型的第1维（仅用数组名）或者剩下维（使用“数组名[表达式]”）的第1维优先被解释为指针。
- 无论被推导变量前面有无“*”，函数名总是解释为指针。

第12章 类型解析、转换与推导

```
#include <stdio.h>     //例12.19
#include <typeinfo>
using namespace std;
int   a[10][20];                          //a的类型为int [10][20]，可理解为 “int (*a)[20]; ”
auto  b(int x) { return x; };             //b的类型为int b(int)
auto  c = a;                              //优先选择int(*c)[20]
auto  *d = a;                             //优先选择int(*d)[20]：和int(*a)[20]匹配的推断结果
auto  e = &a;                             //int (*e)[10][20]：指向数组a
auto  f = printf;                         //  “int (*f)(const char *, …);”
auto  g = a[1];                           //优先选择int *g而非int g[20]
auto  *h = a[1];                          //优先选择int *h而非int h[20]
int   k(int x) { return x; }
void main( ) {
    auto m={ 1,2,3,4 };                   //等价于  “int m[4]= { 1,2,3,4 };”
    auto n= new auto(1);                  //等价于  “int *n=new int(1);”
    auto p = k;                           //p的类型为int (*p)(int)
    auto *q = k;                          //q的类型为int (*q)(int)，*用于匹配k的类型int(*)(int)
```

第12章 类型解析、转换与推导

```
    g[2] = 3;                                   //int *g可当作一维数组int g[ ]使用
    (*p)(4);                                    //调用k(4)
    (*q)(5);                                    //调用k(5)
    printf("%s\n", typeid(a).name( ));          //输出int [10][20]
    printf("%s\n", typeid(a[1]).name( ));       //输出int [20]
    printf("%s\n", typeid(b).name( ));          //输出int __cdecl(int)
    printf("%s\n", typeid(c).name( ));          //输出int (*)[20]
    printf("%s\n", typeid(d).name( ));          //输出int (*)[20]
    printf("%s\n", typeid(e).name( ));          //输出int (*)[10][20]
    printf("%s\n", typeid(f).name( ));          //输出int (__cdecl*)(char const *,...)
    printf("%s\n", typeid(g).name( ));          //输出int *
    printf("%s\n", typeid(h).name( ));          //输出int *
    printf("%s\n", typeid(k).name( ));          //输出int __cdecl(int)
    printf("%s\n", typeid(p).name( ));          //输出int (__cdecl*)(int)
    printf("%s\n", typeid(q).name( ));          //输出int (__cdecl*)(int)
    printf("sizeof(c)=%d\n", sizeof(c));        //输出sizeof(c)=4
}
```

第12章 类型解析、转换与推导

◆12.4 自动类型推导

- **关键字decltype用来提取表达式的类型。**
- **凡是需要类型的地方均可出现decltype。**
- **可用于变量、成员、参数、返回类型的定义以及new、sizeof、异常列表、强制类型转换。**
- **可用于构成新的类型表达式。**

第12章 类型解析、转换与推导

```
int    a[10][20];
decltype(a)* p = &a;  //等价于 “int(*p)[10][20];”  ：a的类型为int [10][20]
decltype(&a[0])h(decltype(a)x, int y) { return x; };          //等价于 “int (*h(int(*x)[20], int))[20];”
//不能定义decltype(a)h(decltype(a)x, int y, int z);          //C++的函数不能返回数组
void sort(double* a, unsigned N, bool(*g)(double, double)) {
    for (int x = 0; x < N - 1; x++)
        for (int y = x + 1; y < N; y++)
            if ((*g)(a[x], a[y])) { double t = a[x]; a[x] = a[y]; a[y] = t; }
}
auto f(double x, double y) throw(const char*)//函数原型为bool (*f)(double, double)
{
    return x > y;
};
```

第12章 类型解析、转换与推导

```
auto g = [ ](int x)->int { return x; };                //在推导g时Lambda表达式被计算并初始化g
decltype(g) (*q)[10];                                  //正确：表达式g的类型已被计算出来
auto r = new decltype(a);                              //等价于int(*r)[20]=new int[10][20];
//decltype([ ](int x)->int{ return x; })*q;            //错误：匿名Lambda表达式未被计算
void main() {
    double a[5];
    decltype(a) *r;                                    //a的类型为double[5]，r的为double (*)[5]
    a[0] = 1;  a[1] = 5;  a[2] = 3;  a[3] = 2;  a[4] = 4;
    sort(a, sizeof(decltype(a)) / sizeof(double), f);
}
```

第12章 类型解析、转换与推导

◆12.5 Lambda表达式

- **Lambda表达式是C++引入的一种匿名函数。**
- **实际上，Lambda表达式被编译为临时类的对象。**
- **该临时类的对象可用于初始化一个变量，此时Lambda表达式被计算。**
- **若未定义存储该临时类对象的变量，则称该Lambda表达式没被计算。**
- **Lambda表达式的声明格式为“[捕获列表](形参列表)mutable 异常说明->返回类型{函数体}”。例如，auto f = [ ](int x=1)->int { return x; };**
- **捕获列表的参数用于捕获Lambda表达式的外部变量。**
- **临时类重载了函数operator()()。当调用f(3)时，等价于f.operator(3)。**

第12章 类型解析、转换与推导

◆捕获列表的参数

- **Lambda表达式的外部变量不能是全局变量或static定义的变量。**
- **Lambda表达式的外部变量不能是类的成员。**
- **Lambda表达式的外部变量可以是函数参数或函数定义的局部自动变量。**
- **出现“&变量名”表示引用外部变量，[&]表示捕获所有函数参数或函数定义的局部自动变量。**
- **出现“=变量名”表示使用外部变量的值（值参传递), [=]表示捕获所有函数参数或函数定义的局部自动变量的值。**
- **参数表后有mutable表示在Lambda表达式可以修改“值参传递的值”，但不影响Lambda表达式外部变量的值。**

第12章 类型解析、转换与推导

```
#include <stdio.h>                //例12.21
#include <typeinfo>
using namespace std;
int main( ) {
    int a = 0;
    auto f = [ ](int x=1)->int { return x; };                    //捕获列表为空，对象f当“准函数”用
    auto g = [ ](int x)throw(int)->int{ return x; };             //g同上：匿名函数抛出异常
    int(*h)(int) = [ ](int x)->int { return x * x; };            //捕获列表为空，h指向“准函数”
    h = f;  //正确：f的Lambda表达式捕获列表为空，f倾向于当“准函数”使用
    auto m = [a](int x)->int { return x * x; };                  //m是“准对象”：捕获a初始化实例成员
    //int(*k)(int)=[a](int x)->int{return x;};                   //函数指针k不能指向准对象（捕获列表非空）
    //h = m;            //错误：m的Lambda表达式捕获列表非空，m倾向于当“准对象”使用
```

第12章 类型解析、转换与推导

```
    //printf(typeid([ ](int x)->int{return x;}).name( ));//错：临时Lambda表达式未计算，无类型
    printf("%s\n", typeid(f).name( ));                    //输出class <lambda_...>
    printf("%s\n", typeid(f(3)).name( ));                 //输出int，使用实参值调用x=3
    printf("%s\n", typeid(f.operator( )( )).name( ));//输出int，使用默认值调用x=1
    printf("%s\n", typeid(f(3)).name( ));                 //输出int
    printf("%s\n", typeid(f.operator( )).name( ));    //输出int __cdecl(int)
    printf("%s\n", typeid(g.operator( )).name( ));  //输出int __cdecl(int)
    printf("%s\n", typeid(h).name( ));                    //输出int (__cdecl*)(int)
    printf("%s\n", typeid(m).name( ));                    //输出class <lambda_...>
    return f(3) + g(3) + (*h)(3);                         //用对象f、g计算Lambda表达式
}//注意：调用g.operator(3)    g(3)
```

第12章 类型解析、转换与推导

◆捕获列表的参数

- 捕获列表的参数可以出现this，但实例函数成员中的Lambda表达式默认捕获this，而静态函数成员中的Lambda表达式不能捕获this。
- 由于this不是变量名或参数名，故不能使用“&this”或者“=this”。
- Lambda表达式的捕获列表非空，倾向于用作“准对象”，否则倾向于用作“准函数”。
- 实例函数成员中的Lambda表达式默认捕获this，故它是一个准对象。
- 只有作为“准函数”才能获得其函数入口地址。

第12章 类型解析、转换与推导

```
int m = 7;                          //全局变量m不用被捕获即可被Lambda表达式使用
static int n = 8;                   //模块变量n不用被捕获即可被Lambda表达式使用
class A {
    int x;                          //由于this默认被捕获，故可访问实例数据成员A::x
    static int y;                   //静态数据成员A::y不用捕获即可被Lambda表达式使用
public:
    A(int m): x(m) { }
    void f(int &a) {                //实例函数成员f()有隐含参数this
        int b = 0;
        static int c=0;             //静态变量c不用被捕获即可被Lambda表达式使用
        auto h = [&, a, b](int u)mutable->int{//this默认被捕获，创建对象h
            a++;        //f()的参数a被捕获并传给h的实例成员a：a++不改变f()的参数a的值
            b++;        //f()的局部变量b被捕获并传给h实例成员b：b++不改局部变量b的值
            c++;        //f()的静态变量c可直接使用，c++改变f()的静态变量c的值
            y=x+m+n+u+c;//this默认被捕获：可访问实例数据成员x
            return a;
        };
```

第12章 类型解析、转换与推导

```
    h(a + 2);                    //实参a+2值参传递给形参u，调用h.operator( )(a+2)
}
static void g(int &a){//静态函数成员g()没有this
    int b = 0;
    static int c = 0;            //静态变量c不用被捕获即可被Lambda表达式使用
    auto h = [&a, b](int u)mutable->int{//没有this被捕获，创建对象h
        a++;                     //g()的参数a被捕获并传给h的引用实例成员a：a++改变参数a的值
        b++;                     //g()的局部变量b被捕获传给h实例成员b：b++不改局部变量b的值
        c++;                     //g()的静态变量c可直接使用，c++改变g()的静态变量c的值
        y=m+n+u+c;               //没有捕获this，不可访问实例数据成员A::x
        return a;
    };
    auto k = [ ](int u) ->int { return u; }; //没有this被捕获，创建对象h
    auto p = k;                              //p的类型为int p(int);
    int (*q)(int) = k;                       //问题：若将红色部分放入void f(int &a)中，如何？
```

第12章 类型解析、转换与推导

```
        h(a + 2);                    //实参a+2值参传递给h形参u
    }
}a(10);
int A::y = 0;                        //静态数据成员必须初始化
void  main( ){
    int p = 2;
    a.f(p);                          //p=2，a.x=10，A::y=30
    a.f(p);                          //p=2，a.x=10，A::y=31
    A::g(p);                         //p=3，a.x=10，A::y=20
    A::g(p);                         //p=4，a.x=10，A::y=22
}
```