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

- 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;       //警告: int -> char，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：自动转换（无警告）
   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;      // 编译时计算 ‘a’ + 1 的值， 没有超过char的范围，不报警
char  v = 'a' + 100;    // 'a’ + 100 超过char的范围（v=-59），报警，不截断
char  w = 300;          // 300超过char的范围（w=44），报警，截断
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，报错
short z = x;            //编译时不可计算，可能截断，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

```
const int  x = 1;
const char  *s1 = "abc";          //data段指针, 指向const段的内容
const char  s2[] = "abc";         //const段数组
void main(void) {
    const int  y = 2;             //stack段变量
    const char  *t1 = "123";      //stack段指针, 指向const段的内容
    const char  t2[] = "123";     //stack数组
    (int &)x = 11;                //error: 修改const段, 程序崩溃
    (char &)s1[0] = '1';          //error: 修改const段, 程序崩溃
    *(char *)(s2+0) = '2';        //error: 修改const段, 程序崩溃
    (int &)y = 22;                //已经修改y存贮单元
    (char &)t1[0] = '1';          //error: 修改const段, 程序崩溃
    *(char *)(t2+0) = '2';
    cout << y << “\n” << t2;      //2, 223, why?
}
```

第12章　类型解析、转换与推导

◆12.1 隐式与显式类型转换

- 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 {
    int  x = 0;
    const int y = 0;
    int q( ) {
        *(int *)&y = y + 1;
        return y;
    }
};
```

```
void main( ) {
    T m;
    const T n;
    int x = m.q( );   // 1
    x = m.q( );       // 2
    x = n.q( );       // error, why?
}
```

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，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，能去除或增加源类型的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为只读全局变量，受保护的区域的全局内存
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，受保护的局部区域
    int w = static_cast<int>(x);        //正确：x有const但被忽略，将只读x的值转换为右值（拷贝）
    //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：转换为左值，但不能去除源类型的属性const
//static_cast<int &>(y) = 0;        //错误：无法去除全局变量y的volatile属性
*(int *)&y = 0;                     //正确：::y = 0
const_cast<int &>(y) = 4;           //正确：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 (实际上已经修改z的存贮单元)
cout << “z=” << z << endl;          //输出z的值，仍然为z=0
*(int *)&z = 3;                     //正确：运行无异常 (实际上已经修改z的存贮单元)
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<T>(左值表达式)”。
- 修改类型的 const 和 volatile 属性，<T> 只能为指针、引用或指向对象成员的指针（可以带 const 和 volatile）。
- 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 num;
public:
    const int nn;
    void dec( ) const;    //const说明对象*this不可写
    Test(int m): nn(m) { num = m; }
};
void Test::dec( ) const {   //this的类型为const Test *const，故Test对象不可修改
    //num--;                                  //错误, *this为const (每个数据成员都是const)
    //nn--;                                   //错误, nn为const, 只读不可写
    //const_cast<Test *>(this)->nn--;    //错误, *this非const, 但其中的nn还是const
    const_cast<Test *>(this)->num--;   //this去除const变为Test *const, 对象可写
    const_cast<int &>(num)--;
    const_cast<int &>(nn)--;
}
```

第12章　类型解析、转换与推导

```
void main( ) {
    Test a(7);                                  //a.num=7，a.nn=7
    a.dec( );                                   //a.num=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
    *const_cast<int *>(&xx) = 2;                //正确，xx = 0？
    a.*const_cast<int Test::*>(&Test::nn) = 3;  //对象实例成员不受保护可修改：a.nn=3
    const_cast<volatile int &>(yy) = 4;         //yy添加volatile，引用yy无内存不受保护，yy=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后不能赋值
    int 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
    //ww = const_cast<volatile int>(xx);        //错误, why?
}
```

第12章　类型解析、转换与推导

◆dynamic_cast——动态转换

- dynamic_cast 在运行时转换：派生类转换为基类、基类转换为派生类。
- dynamic_cast 主要用来解决将基类转换为派生类时的安全问题。
- 格式：dynamic_cast<T> (expr)

  类型T是类的引用、类的指针 或者 void *，

  expr的类型 必须是 类的对象 或者是 类的引用或指针。
- dynamic_cast 转换时不能去除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。
- 将基类转换为派生类时，基类必须包含虚函数或纯虚函数。

第12章　类型解析、转换与推导

```
A  *pa;                                  //A  &pa;
B  *pb = dynamic_cast<B *>(pa);  //B  &pb = dynamic_cast<B &>(pa);
```

(1) 若pa所指的对象没有虚函数，则编译报错；

(2) 若pa所指的对象有虚函数，这时：

(2.1) pa所指的对象与B没有继承关系，则 pb = NULL

(2.2) pa所指的对象是B的祖先类，则 pb = NULL

(2.3) pa所指的对象是B的子孙类，转换正确 (B的父类的指针pa实际上指向B的子孙类)

第12章　类型解析、转换与推导

- 为什么dynamic_cast要求被转换的基类必须有虚函数？

  dynamic_cast 需要知道类的继承关系。对于每个类，编译器会维护一个运行时类型信息 RTTI (RunTime Type Information), dynamic_cast (以及后面的运算符typeid) 从RTTI中获取相关信息。类的继承关系，可以从虚函数表中分析出来。因此，需要基类具有虚函数。

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 A {
    int  m;
    A(int x): m(x) { }
    virtual void f( ) { cout << 'A'; } //若无虚函数, dynamic_cast<B *>(&a) 向下转换出错
};
struct B : public A {                    //A是父类，B是子类
    int  n;
    B(int x, int y): A(x), n(y) { }
    void f( ) { cout << 'B'; }           //函数f()自动成为虚函数
};
void main( ) {
    A a(3);             //b, a.m = 3
    A &b = a;
    B c(5, 7);          //d, c.m = 5,  c.n = 7
    B &d = c;
```

第12章　类型解析、转换与推导

```
B *pc1 = static_cast<B *>(&a);        //语法正确但为不安全的自上向下转换
pc1->f( );                            //输出A，若去除A:f()前面的virtual，结果怎样?
B *pc2 = static_cast<B *>(&b);        //语法正确但为不安全的自上向下转换
pc2->f( );                            //输出A
B *pc3 = dynamic_cast<B *>(&a);  //若a无虚函数f( ), 则转换错误, a不是D的对象导致pc3=0
pc3->f( );                            //运行异常：pc3为nullptr（a非子类对象）
B *pc4 = dynamic_cast<B *>(&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）
A *pb1 = dynamic_cast<B *>(&c);   //语法正确且为安全的自下向上赋值
pb1->f( );                            //输出B：正确的多态行为
A *pb2 = dynamic_cast<B *>(&d);  //语法正确且为安全的自下向上赋值
pb2->f( );                            //输出B：正确的多态行为
B &ra1 = static_cast<B &>(a);         //语法正确但不为安全的自上向下转换
ra1.f( );                             //输出A
```

第12章　类型解析、转换与推导

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

- reinterpret_cast <T> (expr)，将表达式expr转换成不同性质的其他类型T。T类型不能是实例数据成员指针。T 可以是指针、引用、或其他与expr完全不同的类型。
- 将指针转换为足够大的整数，整数类型必须够存储一个地址。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 必须是一个左值表达式。
- 左值引用和右值引用可以相互转换。

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;
```

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, 不能直接写 B &&i = b
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::*q = reinterpret_cast <int B::*> (&B::m);    //&B::m的类型为int B::*，无须转换
    int  kk = reinterpret_cast <int>q;                  //错, 成员指针不能转换、不能参与运算
    q = reinterpret_cast <int B::*> (1);                //错, 成员指针不能转换、不能参与运算
    f = a.*q;                                           //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）只能调用父类定义过的成员函数。
- 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 时需要包含：#include <typeinfo>。

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 A {
    int m;
    A(int x): m(x) {  }
    virtual void f( ) { cout << ‘fa'; }
};
struct B: public A {
    int n;
    B(int x, int y): A(x), n(y) {  }
    void f( )  { cout << ‘fb' << endl; }
    void g( ) { cout << ‘gb' << endl; }
};
```

第12章　类型解析、转换与推导

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

只能:

```
double d = m.operator double() 或 (double)m;
COMPLEX a = m + COMPLEX(2.0, 0.0);
```

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、以及类中用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。
- 数组名代表整个数组类型，函数名代表该函数的指针。
- auto 数组：将 数组类型 的第1维 (最低维) 推导为指针，后面的维类型不变。
- 无论被推导变量前面有无 *，auto将函数名和数组名解释为指针。

第12章　类型解析、转换与推导

```
#include <iostream>
#include <typeinfo>
using namespace std;
int    a[3][4][5];    //可理解为 int (*a)[4][5]
auto  b = &a;         //int (*b)[3][4][5]  (a的类型是int [3][4][5], 取其地址)
auto  *bp = &a;       //int (*bp)[3][4][5] (auto a⇔auto *bp)
auto  c = a;          //int (*c)[4][5] (a的类型是int [3][4][5], 将其第1维推导为指针)
auto  *cp = a;        //int (*cp)[4][5] (auto a⇔auto *cp, a的类型是数组, 所以解释为指针)
auto  d = a[1];       //int (*d)[5] (a[?]的类型是int [4][5], 将其第1维推导为指针)
auto  *dp = a[1];           //int (*dp)[5]  (auto d⇔auto *dp)
auto  e = a[1][2];          //int *e  (a[?][?]的类型是int [5], 将其第1维推导为指针)
auto  *ep = a[1][2];        //int *ep (auto e⇔auto *ep)
auto  f = a[1][1][1];       //int g
auto  *fp = a[1][1][1];   //error (a[?][?][?]的类型是int, 不是数组, 因此与auto *fp不等价)
auto  h(int x) { return x; };  //int h(int)
auto  g = printf;                   //int (*g)(const char *, …)
```

第12章　类型解析、转换与推导

```
int main( ) {
    auto m = { 1, 2, 3 };       //int m[3] = { 1, 2, 3 }
    auto n = new auto(1);       //int *n = new int(1)
    auto p = h;                 //int (*p)(int)  (函数名总是解释为指针)
    auto *q = h;                //int (*q)(int)  (h是函数, 因此 auto p ⇔ auto *q)
    (*p)(4);                    //调用 b(4)
    (*q)(5);                    //调用 b(5)

    cout << typeid(a).name();       //int [3][4][5]  (注意 auto  c = a 中c的类型)
    cout << typeid(a[1]).name();  //int [4][5]      (注意 auto  d = a[1] 中d的类型)
    cout << typeid(d).name();      //int (*)[5]
    cout << typeid(p).name();      //int (_cdecl *)(int)
    cout << 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];
auto r = new decltype(a);    //int (*r)[20] = new int[10][20];  //auto与decltype的区别?
decltype(a) *p = &a;          //int (*p)[10][20]
decltype(&a[0]) h(decltype(a) x, int y) { return x; };   //int (*h(int x[10][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2维数组
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)    //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; };        //匿名对象被g创建, 已经知道类型
decltype(g) (*q)[10];                          //正确：表达式g的类型已被计算出来
decltype([ ](int x)->int { return x; }) *q;   //错误：匿名对象未被创建, 不知道类型

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表达式是一个匿名函数，该函数实现了一个匿名类，匿名类中主要包括匿名的构造函数和 ( )运算符重载函数 operator( )。**
- **lambda表达式的函数体就是 operator( )。调用 lambda表达式实际上就是调用 ( )运算符的重载函数 operator( )。**
- **operator() 是 const 的，即 operator( )(…) const，因此不能修改匿名类内的任何实例数据成员。**
- **定义一个lambda表达式，则创建一个匿名类，同时创建该匿名类的一个对象。**

第12章　类型解析、转换与推导

lambda表达式形式：

[capture list] (parameter list) -> return type { function body }

capture list： 捕获列表，用于获得lambda函数体外变量的值（lambda表达式根据这些捕获到的变量创建匿名类的同名实例成员变量）。捕获可以分为按值捕获和按引用捕获。非局部变量，如静态变量、全局变量等不需要捕获，直接使用。

parameter list： 参数列表（调用( )运算符函数时传入的参数），可以省略。从C++14开始，支持默认参数。

return type： 返回值类型。可以省略，这种情况下根据lambda函数体中的return语句推断出返回类型，如果函数体中没有return，则返回类型为void。

function body： 函数体（即 ( )运算符的重载函数 operator( )(...) 的函数体）。

第12章　类型解析、转换与推导

Lambda表达式的调用方式:

auto f = [ ](int x)->int { return x * x; };　//创建一个匿名类, 同时创建对象f.

int x = f(10);　//等价：int x = f.operator()(10)　**( ) 运算符的调用方式 ???**

捕捉变量：捕捉lambda函数体外变量的值（lambda表达式根据这些捕获到的变量创建匿名类的实例成员变量）。

[ ] 不捕获任何变量。

[&] 以引用方式捕获所有变量（可以修改匿名类变量的值）。

[=] 用值的方式捕获所有变量（不能修改匿名类变量的值）。

[varName] 以值方式捕获变量varName（不能修改匿名类内varName的值）。

[&varName] 以引用方式捕获变量varName（可以修改varName的值）。

[this] 捕获所在类的this指针。

第12章　类型解析、转换与推导

Lambda表达式的本质：

```
int a = 1;
int main() {
    static int b = 2;
    int  m = 3, n = 4;
    char *s = new char [10] {'a', 'b', 'c', 0};
    auto f = [m, &n, s](int x)->char * {
        s[0] += m+n+x+a+b;
        //m++; //错：匿名类的实例数据成员m是const的
        n++;  //对：匿名类的实例数据成员n是引用变量,
              //      可以修改所指向的内存单元
        a++;  //对：全局变量a不是匿名类的数据成员
        b++;  //对：静态变量b不是匿名类的数据成员
        return s;
    }; //创建匿名类及其对象f
    f(5)[0] = '1';       //等价：f.operator()(5)[0] = '1'
    std::cout << s;     //1bc
    f.operator()(6);    //等价：f(6)
    std::cout << s;     //Dbc
}
```

Lambda表达式的解释：

为了方便解释，下面用 A 表示匿名类的名称。

调用 f(…) ⇔ f.operator()(…)

```
class A {
    int  m, &n;
    char *s;
    A(int m, int &n, char *s): m(m), n(n), s(s) { }
    char *operator()(int x) const {
        (A::s)[0] += A::m + A::n + x + a + b;
        //A::m++;  //错, 不能改变A::m
        A::n++;
        ::a++;
        b++;    //main::b++
        return A::s;
    }
} f;
```

第12章　类型解析、转换与推导

Lambda表达式的调用机制：

- 定义Lambda表达式及其对象时，将创建一个匿名类，同时创建一个该对象。
- 每次调用 Lambda表达式，都是利用该对象去调用（）运算符重载函数operator()，即：对象.operator()(…)，也可写成：对象(…)。
- operator()的属性是const的，因此 operator()不能修改匿名类中的非引用类型的实例成员变量。但通过 将 Lambda 表达式修改为 mutable 属性，使得匿名类中的所有实例成员变量都具备 mutable 属性，这样 operator() 可以修改匿名类中所有的实例成员变量。

第12章　类型解析、转换与推导

Lambda表达式的调用机制解释:

```
int main() {
    static int a = 1;
    int  m = 2;
    auto f = [m](int x) mutable -> int {
        m += a + x;
        return m;
    }; //创建匿名类及其对象f
    int  i = f(0);    //i = 3 （ f.operator()(0) ）
    int  j = f(0);    //j = 4
    printf("%d, %d, %d \n", m, i, j);  //2, 3, 4
}
```

对Lambda表达式的解释:

为了方便解释，下面用 A 表示匿名类的名称。

f (…) ⇔ f.operator()(…)

```
class A {
    mutable int  m;
    A(int m): m(m) { }
    int  operator( )(int x) const {
        A::m += a + x;   // A::m += main::a + x
        return A::m;
    }
} f;
```

第12章　类型解析、转换与推导

```
#include <iostream>
int a = 1;
int main( ) {
    static int x = 3;
    int y = 4;
    int z = 5;
    auto f = [y, &z](int v) -> int {
        //y++;    //错：f是const, 不能修改匿名类的成员变量 y
        z++;      //对：z是引用, 可以修改引用所指的变量
        return a+x+y+z+v;
    }; //创建一个匿名类及其对象f
    auto g = [=](int v) -> int {
        //z++;   //错：  g是const, 不能修改匿名类的成员变量 z
        return a+x+y+z+v;
    }; //创建一个匿名类及其对象g
    auto h = [=](int v) mutable -> int {
        y++;    //对：  mutable int y
        z++;    //对：  mutable int z
        return a+x+y+z+v;
    }; //创建一个匿名类及其对象h
    int z1 = f(100);    // z1 = ?
    int z2 = g(100);    // z2 = ?
    int z3 = h(100);    // z3 = ?
}
```

z1 = 114

z2 = 113

z3 = 115

如果将 int z1 = f(100) 移到 auto g = [=] 之前，z1、z2、z3 的值又是多少？

z1 = 114

z2 = 114

z3 = 116

第12章　类型解析、转换与推导

- **Lambda表达式的匿名类与普通匿名类的区别**

  - 普通的匿名类可以生成多个有名字的对象，Lambda表达式的匿名类只能产生一个有名字的对象（这个对象是在定义Lambda表达式时创建的）；
  - 普通匿名类的实例成员函数有对象的this指针，Lambda表达式匿名类的实例成员函数没有匿名类对象的 this 指针.

第12章　类型解析、转换与推导

- 可以用函数指针指向捕获列表为空的Lambda表达式的 ( ) 函数.

```
auto f = [ ](int x)->int { return x * x; };
auto g = [ y ](int x)->int { return x + y; };
int (*p)(int) = f;   //对，p 指向匿名类的函数 operator()(int x)
int  z = p(10);      //z = 100
int (*q)(int) = g;   //错，g的捕获列表不为空
```

(1) 这里 f 和 g 的类型是2个匿名类，而 p 和 q 是2个普通函数的指针。

(2) p的类型不能auto，即不能 auto (*p)(int) = f，因为 f的类型是匿名类，编译器不会将 f.operator()(int x) 的返回值类型推断给 p, auto推断出的是匿名类。下面语句是合法的:

```
auto p = f;  auto *p = &f;  decltype(f) *p;  //推断出的是匿名类
```

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; }; //错误：函数指针只能指向捕获列表为空的准对象
    //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.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

```
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;           //静态变量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 Lambda 表达式（类）
    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
}
```