



华中科技大学

第12章 类型解析、转换与推导

许向阳

xuxy@hust.edu.cn





第12章 类型解析、转换与推导

12.1 隐式与显式类型转换

12.2 cast系列类型转换

12.3 类型转换实例

12.4 自动类型推导

12.5 lambda表达式





12.1 隐式与显式类型转换

隐式类型转换

- 定义变量，给变量初始化时

```
int x= '1' ;    // x=49;
```

- 赋值时

```
char c='2';      x = c;
```

- 运算时

```
x+= '3';
```

- 函数调用时

参数传递、返回结果的传递





12.1 隐式与显式类型转换

函数调用时，参数类型的隐式转换

```
double area(double r)
{
    return 3.14*r*r;    // 浮点常量3.14默认其为double类型
}

void main( )
{
    char m=0x1234;    // 警告：“初始化”截断常量值
                     // 截断后 m=0x34;
    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和实参x的类型相容，可自动转换
    a=area('A');       // 字符'A'转换为double类型，类型相容
    cout<<"Area="<<a;
}
```





12.1 隐式与显式类型转换

隐式类型转换

```
class complex { public: int r,c; ...};
```

```
class D : public complex {...};
```

```
complex *a ;          D d(...), *p;
```

```
a = &d;                // 赋值时，隐式进行类型转换
```

```
// 类似的还有： a = new D(...);    a = p;
```

➤ 由单参数的构造函数进行类型转换；

complex(int r, int i=0) { ... }, 可以将一个数转换为
complex类型, $2 \Leftrightarrow \text{complex}(2)$

➤ 在运算符重载中，重载类型转换函数

```
operator int( ) { return r+i; }
```





12.1 隐式与显式类型转换

显式类型转换

```
int x;    double y;
```

```
x=int(y);
```

```
char buf[1024];
```

```
x=*(int *)(buf+10);
```

```
class myclass {.....};
```

```
myclass a(...);
```

```
a=*(myclass *)(buf+10);
```





12.1 隐式与显式类型转换

- 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左值引用转换，则转换结果为左值；

```
int x=0;
```

```
int(y)=10;    // 等同 int y=10;
```

```
(short) x=1;  //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 =2;    // 报错：传统右值不能出现在等号的左边  
              //      尽管 x 转换前也是 int
```

```
(int &)x=3;    // 正确，是引用类型int &
```

```
*(int *)&x =4; //正确
```





12.1 隐式与显式类型转换

- 注意 **const** 变量的类型转换

对于初值为常量的 **const** 变量，类型转换的语句无效。

```
const int x=0;  
int    y;  
const int  z=y;
```

```
(int &)x=20;      ⇔      *(int *)&x=20;  
cout<<x<<endl;    // x=0
```

```
(int &)z=20;      ⇔      *(int *)&z =20;  
cout<<z<<endl;    // z=20
```





華中科技大學

12.1 隐式与显式类型转换

地址类型转换 VS 数据类型转换





12.1 隐式与显式类型转换



```
int xxx = 10;
char pa[10]="012345";
char* pc;
xxx =(int) pa;
xxx = *pa;
xxx = *(int*)pa;

pc = pa;
pc = (char *)xxx;
pc = (char*)&xxx;
```

0x30
0x31
0x32
0x33
0x34
0x35
00

0x0a
00
00 00

pc 0x00a3fe78
pa 0x00a3fe84

运行后xxx中的值
0x00a3fe84
0x00000030
0x33323130

xxx 0x00a3fe98

pc 中的值，依次是 0x00a3fe84, 0x33323130, 0x00a3fe98





12.1 隐式与显式类型转换

- 无风险的转换由编译程序自动完成，不给程序员提示，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。





12.1 隐式与显式类型转换

- X86编译模式简单类型字节数:

$\text{sizeof}(\text{bool}) \leq \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int})$
 $\leq \text{long} \leq \text{float} \leq \text{double} \leq \text{long double} .$

- 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。



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。

▲ C/C++	使用 Windows 运行时扩展	
常规	取消显示启动版权标志	是 (/nologo)
优化	警告等级	等级 3 (/W3)
预处理器	将警告视为错误	否 (/WX-)
代码生成	警告版本	





12.2 cast系列类型转换

- C++ 中，继续支持 C 语言的强制类型转换；
- C++中，引入 4个强制类型转换关键字

静态转换 `static_cast` 只读转换 `const_cast`

动态转换 `dynamic_cast` 重释转换 `reinterpret_cast`

皆为显式类型转换，对类型转换进行更严格的限制。

.....`_cast`<目标类型表达式>（源类型表达式）

```
*const_cast<int *>(&ppp) = 20;
```

```
*(int *)&ppp = 20;
```





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`volatile`属性，不做多态相关的检查；编译期检查类型能否转换；
- `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`volatile`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`主要用于将源表达式重新解释为一种新的类型。





12.2 cast 系列 类型转换

static_cast<T>(expr)

不是所有类型都能转换的

int x; double y; float z;

x= static_cast<int>(y); ⇔ x=int(y);

static_cast<int>(y) = x; // error : =的左操作数必须是左值

***static_cast<int *>(&y)=x;** // 无法从double * 转换为 int *

***static_cast<int *>(&z)=x;** // 无法从float * 转换为 int *

讨论： 运行结果的分析, Why?

x=10; y=5.2;

(int)&y = x; // cout<<y; 显示 5.2

*(int *)&z =x; // cout<<z; 显示 1.4013e-44



y

5.200000000000000002



y

5.1999969482421964





12.2 cast系列类型转换

static_cast<T>(expr)

- 目标类型不能包含存储位置类修饰符，如static、extern、auto、register等；
- 仅在编译时静态检查源类型能否转换为目标类型，运行时不做动态类型检查；
- 不能去除源类型的const或volatile；
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)。





12.2 cast系列类型转换

const_cast<目标类型>(源类型)

- 目标类型必须是指针、引用、或者指向对象成员的指针；
- 类型表达式不能包含存储位置类修饰符，如static、extern、auto、register等；
- 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



12.2 cast系列类型转换

```
const int ppp = 0;
```

```
int qq = 10;
```

```
static_cast<int&>(qq) = 20;
```

```
static_cast<int&>(ppp) = 20;
```

//无法从const int转为 int &

```
const_cast<int&>(ppp) = 20;
```

```
*const_cast<int*>(&ppp) = 20;
```

```
*static_cast<int*>(&ppp) = 20;
```

//无法从const int * 转为 int *





12.2 cast系列类型转换

dynamic_cast<T> (expr)

- 类型T是类的引用、类的指针 或者 void*类型;
- expr的源类型必须是类的对象、对象地址
- 子类向父类转换
- 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;
- 转换时不能去除源类型中的const和volatile属性;
- 有址引用和无址引用之间不能相互转换;

基类指针转换为派生类指针，基类必须包含虚函数或纯虚函数，确保基类对象就是派生类对象（可用typeid检查）





12.2 cast系列类型转换

```
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  
    // 编译报错: 基类无多态性  
};
```

```
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  
};
```





12.2 cast系列类型转换

```
B a(3);          B &b=a;  
D c(5,7);        D &d=c;  
D *pc1 = static_cast<D*>(&a); // 不安全的自上向下转换  
pc1->f( );        // 输出B
```

讨论：自上（基类）向下（派生类）转换，为什么不安全？

```
pc1-> n =20;
```

pc1是 D类的指针，可用以访问 D类的成员；

但实际上，pc1指向的是 B类对象的空间，并没有D类的成员。运行后，系统会崩溃。





12.2 cast系列类型转换

B a(3); B &b=a; B *pb;

D c(5,7); D &d=c;

D *pc1 = **static_cast<D*>(&a);** // 不安全的自上向下转换

D *pc3 = **dynamic_cast<D*>(&a);**

// 基类向派生类转换，若基类无虚函数f()，编译报错：

// 基类不是多态类型！**操作数必须包含多态类型**

➤ 若是由 派生类向基类转换，则不需要类型的多态性。

B *pb = **dynamic_cast<B*>(&c);**

类型由下（派生类）向上（基类）转换，总是安全的。





12.2 cast系列类型转换

```
B a(3);      B &b=a;   B *pb;
```

```
D c(5,7);    D &d=c;
```

```
D *pc3 = dynamic_cast<D*>(&a);
```

```
pc3->f();    // 运行时异常: pc3为nullptr (a非子类对象)
```

Question : 基类指针向派生类指针转化,
运行时, 是否一定会出现异常?

Answer: 不一定。

```
D *pc3 = dynamic_cast<D*>(pb);
```

pb 虽然是基类指针, 但它指向的对象可以是基类;
如 pb=&a; 也可以指向派生类, 如 pb =&c;





12.2 cast系列类型转换

D *pc3 = dynamic_cast<D*>(&a);

push 0

push offset D `RTTI Type Descriptor' (061C14Ch)

push offset B `RTTI Type Descriptor' (061C138h)

push 0

lea eax,[a]

push eax

call ____RTDynamicCast (0611410h)

add esp, 14h // 20个字节的参数

mov dword ptr [pc3],eax

在程序运行的过程中，进行检查，运行后 pc3=NULL;
即隐含不能自上向下转换





12.2 cast系列类型转换

跳过

`D *pc2 = static_cast<D*>(&b);` //不安全的自上向下转换

`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);` //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);` //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);` //不安全,自上向下转换

`ra1.f();` //输出B



12.2 cast系列类型转换

跳过

左值引用的类型转换

`D &ra2 = static_cast<D&>(b);` //不安全，自上向下转换

`ra2.f();` //输出B:

`B &rc1 = dynamic_cast<D&>(c);` //安全，自下向上赋值

`rc1.f();` //输出D: 正确的多态行为

`B &rc2 = dynamic_cast<D&>(d);` //安全的，自下向上赋值

`rc2.f();` //输出D: 正确的多态行为



12.2 cast系列类型转换

跳过

右值引用的类型转换

B &&rc3 = static_cast<D&&>(c); //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); //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); //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





12.2 cast系列类型转换

reinterpret_cast

- 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
- 指针与足够大的整数类型之间的相互转换；
- 所谓足够大的整数类型：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。





12.2 cast 系列 类型转换

```
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;  
  
B a(1);  
B &b = a;    //b有址引用a, 共享a的内存
```





12.2 cast系列类型转换

B a(1) , *e;

e = &a;

e = static_cast<B*>(&a);

e = const_cast<B*>(&a);

e= dynamic_cast <B*> (&a);

e = reinterpret_cast <B*> (&a);

5条语句完全等同, 实际上 都没有类型转换

= 左右都是 B * 类型

mov dword ptr [e], offset a





12.2 cast系列类型转换

```
int f;          B *e;  
f = e;          // 无法从 B* 转换为 int  
f = static_cast <int> (e); // 无法从 B* 转换为 int  
f = const_cast <int> (e); // 无法从 B* 转换为 int  
f = dynamic_cast <int> (e); // 目标类型无效  
                        // 目标类型要求是类的指针、引用
```

```
f = reinterpret_cast <int> (e); //指针e转为整型，赋给f
```

```
f = int(e);    // 等同 reinterpret_cast
```

```
mov    eax,dword ptr [e]
```

```
mov    dword ptr [f],eax
```





12.2 cast系列类型转换

跳过

```
int 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
```



华中科技大学

跳过

12.2 cast系列类型转换

```
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
```





12.2 cast系列类型转换

跳过

```
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
```





12.2 cast 系列 类型转换

跳过

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



12.3 类型转换实例

- 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)。





12.3 类型转换实例

- 关键字 **typeid** 可以获得对象的真实类型标识
- 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
- typeid 是关键字，并不是函数；
- typeid 的结果有时候在编译期确定，有时在执行期确定；
- 在使用 typeid 之前可先 “#include <typeinfo>”，在 std 名字空间。





12.3 类型转换实例

- `type_info` 是一个类
- 不同的编译器实现的 `type_info` 类可能不同
- C++ 要求 必须实现 `name()`,
其返回类型是 `const char *`,
- 有 `==`、`!=`、`before`、`raw_name`、`hash_code`等函数。
- 详细信息见头文件 `<typeinfo>`



12.3 类型转换实例

```
#include <typeinfo>
const type_info & ti = typeid(int);
cout<<ti.name()<<endl;
cout<<ti.raw_name()<<endl;
cout<<ti.hash_code()<<endl;
```

```
int x = 20;
const type_info& tx = typeid(x);
cout << tx.name() << endl;
cout << tx.raw_name() << endl;
cout << tx.hash_code() << endl;
```

```
C:\教学\本科教学\面向
int
.H
3440116983
int
.H
3440116983
```

typeid使用格式:

typeid(类型表达式)

typeid(数值表达式);





12.3 类型转换实例

```
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; }  
};
```





12.3 类型转换实例

```
B a(3);                B* pb = &a;
D c(5, 7);
D* pc(nullptr);        //定义子类指针pc并设为空指针

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);
    pc = dynamic_cast<D*>(pb);    // B须有虚函数
    pc = reinterpret_cast<D*>(pb);
    pc->g();            //输出G，不转换pb无法调用g()
}
```

Q： if 条件是否成立？
若在if 语句前有， `pb = &c;` 结果如何？





12.3 类型转换实例

```
B  a(3);                B* pb = &a;
D  c(5, 7);
D* pc(nullptr);
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
cout << typeid(*pc).raw_name() << endl;
cout << typeid(*pc).hash_code() << endl;
```





12.3 类型转换实例

RTTI: Run Time Type Identification / Information
运行时类型信息

编译器会在每个含有虚函数的类的虚函数表的前四个字节存放一个指向_RTTICompleteObjectLocator结构的指针。

_RTTICompleteObjectLocator就是实现dynamic_cast的关键结构。

存放了vfptr相对this指针的偏移，构造函数偏移（针对虚拟继承），type_info指针，以及类层次结构中其它类的相关信息。如果是多重继承，这些信息更加复杂。





12.3 类型转换实例

要求显示调用类型转换函数

在函数名前加 关键字: **explicit**

- **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。





12.3 类型转换实例

```
class COMPLEX {  
    double r, v;  
public:  
    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);    };  
    operator double() { return r; }  
}m(2,3);  
  
double d=m;    // d=m.operator double( );  
               // d=double(m);  
m+2.0 等价于 m+COMPLEX(2.0, 0.0)
```





12.3 类型转换实例

```
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);  
  
double d=m; //error无法从COMPLEX 转换为 double  
z=m + 2.0; // error :没有与操作数匹配的运算符  
  
double d=m.operator double( );  
        d = double(m);  
z=m + COMPLEX(2.0);
```





12.4 自动类型推导

C 语言程序中（文件命名形如 test.c）

关键字 **auto** 作为类型修饰符

```
auto int x=10;           // int x=10;
```

```
auto char p[] = "abcd"; // char p[]="abcd";
```

- 函数内部定义的变量，缺省类型为 auto;
- 用于定义函数内部的局部、非静态变量;
- 局部非静态变量的空间分配在栈上;
- 函数返回时，变量的空间自动回收;
- 函数的形参也是自动变量。





12.4 自动类型推导

C++ 语言程序中（文件命名形如 test.cpp）

关键字 **auto** 用于**类型的自动推导**

```
auto x=10;           // int x=10;
```

```
auto y = 3.5;        // double y = 3.5;
```

```
auto p = "abcd";     // const char *p="abcd";
```

```
auto z;             // error, 无法推导 z 的类型
```

```
auto int w =10;    // error, 类型说明符的组合无效
```

```
auto q[]="abcd";    // auto不能出现在顶级数组类型中
```





12.4 自动类型推导

在C++中，**auto**用于类型推导

- 可用于推导变量、各种函数的返回值；
- 不能用于函数形参的推导；
- 不能用于类中一般数据成员的类型推导，
但可用于 `const static`（静态常量）的成员类型推导；
- 可用于全局变量、局部变量的推导；
- 被推导实体不能出现类型说明；
- 被推导实体可以出现存储可变特性 `const`、`volatile`；
可以出现存储位置特性，如 `static`、`register`。





12.4 自动类型推导

与数组有关的自动推导

```
int ta[4] = { 10,20,30,40 };
```

```
auto tb = ta;    // tb 的类型为 int *
```

```
    lea    eax, [ta]
```

```
    mov    dword ptr [tb], eax
```

```
*tb = 12;        // ta[0]=12;
```

```
*(tb + 1) = 15;  // ta[1]=15;
```





12.4 自动类型推导

与数组有关的自动推导

```
int ta2[2][4] = { { 10,20,30,40 }, { 21,22,23,24 } };  
auto tb2 = ta2;    // tb2 的类型是 int [4] *  
                  // 等同于 int (*tb2)[4] = ta2;  
  
*tb2[0] = 12;      // ta2[0][0] = 12;  
  
*(tb2 + 1)[0] = 15; // ta2[1][0] = 15;  
  
tb2 +=1;           // tb2指向数组的下一行,  
                  // 实际值增加 16个字节
```





12.4 自动类型推导

与数组有关的自动推导

```
int ta2[2][4] = { { 10,20,30,40 }, { 21,22,23,24 } };
```

```
auto tb2 = ta2;    // tb2 的类型是 int [4] *
```

```
                // 等同于 int (*tb2)[4] = ta2;
```

```
auto* tp1 = ta2;   // int [4] *
```

```
auto* tp2 = &ta2;  // int [2][4] *
```

```
                // int (*tp2)[2][4] = &ta2;
```





12.4 自动类型推导

为什么要类型推导？

```
#include <map>
std::map<double, double> resultMap;

std::map<double, double>::iterator it = resultMap.begin();
```

更简单的写法：

```
auto it = resultMap.begin();
```





12.4 自动类型推导

为什么要类型推导？

```
#include <map>
int key;
std::multimap<int, int> resultMap;
std::pair< std::multimap<int, int>::iterator,
          std::multimap<int, int>::iterator >
range = resultMap.equal_range(key);
```

更简单的写法：

```
auto range = resultMap.equal_range(key);
```





12.4 自动类型推导

表达式类型的提取

- 关键字 `decltype` 用来提取表达式的类型;
`decltype(表达式)`
- 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;
- 可用于 `new`、`sizeof`、异常列表、强制类型转换;
- 可用于构成新的类型表达式。





12.4 自动类型推导

```
int x = 10;
```

```
decltype(x) y;    // int y;
```

```
int a1[10];
```

```
decltype(a1) p1;    // int p1[10]; p1的类型是 int[10]
```

```
decltype(a1) *p11; // int (*p11)[10]; int[10] *
```

```
p1[2] = 12;
```

```
p11 = &a1;
```

```
(*p11)[2] = 12;    // (*&a1)[2] = a1[2] = 12;
```





12.4 自动类型推导

```
int a2[10][20];  
decltype(a2) p2;    // int p2[10][20]; 类型是 int [10][20]  
decltype(a2)* p22;  // int (*p22)[10][20];  
                    // 类型是 int[10][20] *  
  
p22 = &a2;  
p2[0][1] = 11;  
(*p22)[0][1] = 12; // (*&a2)[0][1] = a2[0][1]
```

12.5 Lambda表达式



华中科技大学



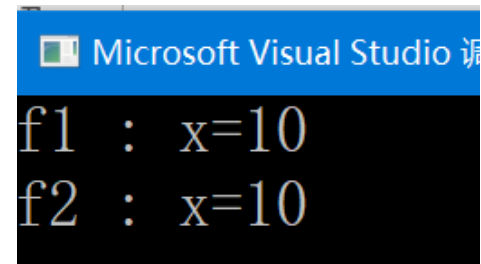


12.5 Lambda表达式

```
#include <iostream>
using namespace std;
void f1(int x)
{    cout << "f1 : x=" << x << endl; }
```

```
int main()
{    auto f2 = [](int x) {cout << "f2 : x=" << x << endl;};
    f1(10);
    f2(10);
    return 0;
}
```

- **f1**是一个函数，是全局性的一个符号；
- **f2**是一个表达式，是一个局部符号，只能在定义它的函数内部使用；
 f2 类似一个函数，匿名的函数；
- 在一个函数内部，是不能再定义函数的。





12.5 Lambda表达式

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

```
int f1(int x, int y)
{   return x + y; }
```

```
int main()
{
```

```
    auto f2 = [](int x, int y) ->int {return x + y;};
    cout << "f1(10,20)= " << f1(10,20) << endl;
    cout << "f2(10,20)= " << f2(10,20) << endl;
    return 0;
}
```

➤ **Lambda 表达式 有返回类型**

➤ **用 lambda 表达式 取代了函数，表达更简洁**

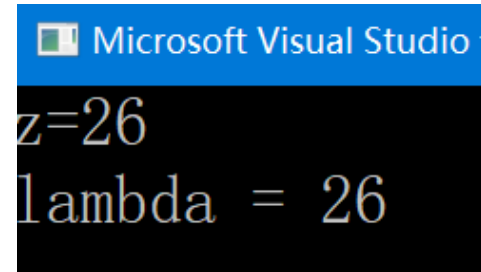
Microsoft Visual Studio 调试

```
f1(10, 20) = 30
f2(10, 20) = 30
```



12.5 Lambda表达式

```
class A {  
private: int x, y;  
public:  
    A(int x, int y):x(x), y(y) { }  
    int operator ()(int a, int b) { return x * y + a * b; }  
};  
int main()  
{  
    A temp(2, 3);  
    int z = temp(4, 5); // int z = temp.operator ()(4, 5);  
    cout << "z=" << z << endl;  
    int u = 2, v = 3;  
    auto la = [u, v](int a, int b)->int {return u * v + a * b;};  
    cout << "lambda = " << la(4, 5) << endl;  
    return 0;  
}
```



- Lambda表达式实现匿名类
- 用匿名类来代替有名类，表达更简洁





12.5 Lambda表达式

```
class A {  
private: int x, y;  
public:  
    A(int x, int y):x(x), y(y) { }  
    int operator ()(int a, int b) { return x * y + a * b; }  
};  
int main()  
{  
    int u = 2, v = 3;  
    A temp(u, v);  
    int z = temp(4, 5); // int z = temp.operator ()(4, 5);  
    cout << "z=" << z << endl;  
    auto la = [u, v](int a, int b)->int {return u * v + a * b;};  
    cout << "lambda 1= " << la(4, 5) << endl;  
    u = 10; v = 20;  
    cout << "lambda 2 = " << la(4, 5) << endl;  
    z = temp(4, 5);  
    cout << "z=" << z << endl;  
    return 0;  
}
```

```
z=26  
lambda 1= 26  
lambda 2 = 26  
z=26
```

➤ 分析：为什么修改 u, v 后，输出结果不变？





12.5 Lambda表达式

```
void f()
{
    int u = 2, v = 3;
    auto la = [&u, &v](int a, int b)->int {return u * v + a * b;};
    cout << "lambda 1= " << la(4, 5) << endl;
    u = 10; v = 20;
    cout << "lambda 2 = " << la(4, 5) << endl;
}
```

Microsoft Visual Studio 调试控

```
lambda 1= 26
lambda 2 = 220
```

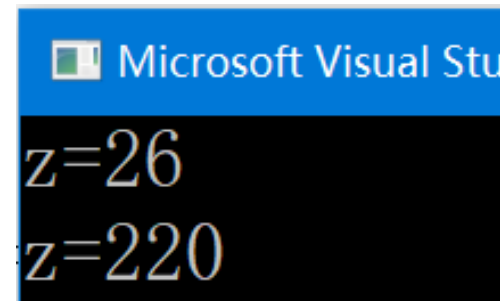
- 分析：为什么将 [u, v] 改为 [&u,&v]后，输出结果变化？
- 对应的类如何写？



12.5 Lambda表达式

```
class B {  
private: int &x, &y;  
public:  
    B(int &x, int &y) :x(x), y(y) { }  
    int operator ()(int a, int b) { return x * y + a * b; }  
};
```

```
void f1()  
{  
    int u = 2, v = 3;  
    B temp(u, v);  
    int z = temp(4, 5);  
    cout << "z=" << z << endl;  
    u = 10;    v = 20;  
    z = temp(4, 5);  
    cout << "z=" << z << endl;  
}
```



Microsoft Visual Studio

z=26

z=220

➤ Lambda表达式翻译成类





12.5 Lambda表达式

- Lambda表达式是C++引入的一种匿名函数
- Lambda表达式的声明格式

[捕获列表](形参列表)mutable 异常说明->返回类型{函数体}

[capture list](params list) mutable exception -> return type {
function body }

capture list: 捕获外部变量列表

mutable: 用于说明是否可以修改捕获变量
有mutable, 则可以修改捕获变量

exception: 异常接口说明





12.5 Lambda表达式

- Lambda表达式是C++引入的一种匿名函数
- Lambda表达式的声明格式

[捕获列表](形参列表)mutable 异常说明->返回类型{函数体}

[捕获列表](形参列表) ->返回类型{函数体}

[捕获列表](形参列表) {函数体}

[捕获列表]{函数体}





12.5 Lambda表达式

► Lambda 表达式实现的机理

```
zz = 0;
```

```
008420D0 C7 45 DC 00 00 00 00 mov      dword ptr [zz], 0
auto surplus = [](int sal, int exp) ->int {return sal - exp;};
008420D7 33 C0                      xor      eax, eax
008420D9 88 85 07 FF FF FF          mov      byte ptr [ebp-0F9h], al
```

```
zz = surplus(100, 80);
```

```
008420DF 6A 50                      push     50h
008420E1 6A 64                      push     64h
008420E3 8D 4D D3                  lea      ecx, [surplus]
008420E6 E8 85 F9 FF FF          call     <lambda_87ce62b9f68fc533a39a7753f0c263f7>::operator() (0841A70h)
008420EB 89 45 DC                  mov      dword ptr [zz], eax
```





12.5 Lambda表达式

Lambda 表达式的实现原理

```
class lambda_xxxx {  
private:  
    int a;    int b;  
public:  
    lambda_xxxx(int _a, int _b) :a(_a), b(_b) { }  
    int operator( )(int x, int y) throw ()  
    {  
        return a + b + x + y ; }  
};
```

mutable: 用于说明是否可以修改捕获变量
如果lambda 表达式中，没有mutable, 就相当于
int operator()(int x, int y) **const** ...





12.5 Lambda表达式

准确理解 常成员函数

```
class lambda_yyyy {  
private:  
    int &x;          int y;  
    char *p;  
public:  
    lambda_yyyy(int &_x, int _y) :x(_x) { ..... }  
    int operator( ) ( ) const  
    {   y=20;        // 错误: 不可修改 y  
        x+=20;  
        p = new char[10]; // 错误: 不可修改 p  
        *p = 'A' ;  
        return 10;  
    }  
}; int const y;  char * const p; int & const x;
```





12.5 Lambda表达式

- Lambda 表达式的本质就是重载了()运算符的类
- 这种类通常被称为 functor, 即行为像函数的类
- lambda 表达式对象其实就是一个匿名的 functor
- 定义表达式时, 就是定义一个类的对象
- 捕获列表中的变量, 是对象构造函数的参数
- 在使用表达式时, 捕获列表中的参数是不会传递的;
除非使用的是参数的引用, “间接”使用了参数的最新值。





12.5 Lambda表达式

- **捕获列表**的参数用于捕获Lambda表达式的外部变量；
- 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；
- [&] 捕获“**引用**”**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；
- 外部变量不能是全局变量或static定义的变量；
- 外部变量不能是类的成员；
- 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。





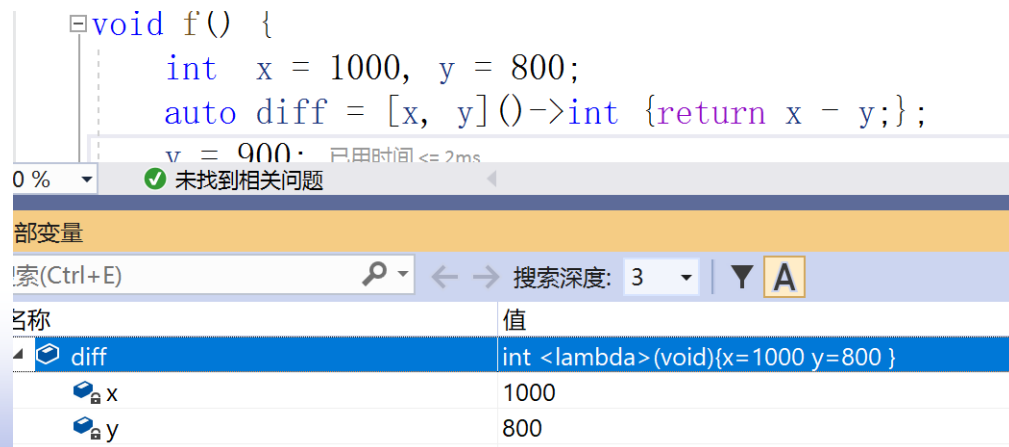
华中科技大学

12.5 Lambda表达式

[捕获列表](形参列表)mutable 异常说明->返回类型{函数体}

```
void f() {  
    int x = 1000, y = 800;  
    auto diff = [x, y]() -> int {return x - y;};  
    y = 900;  
    cout << diff() << endl; // 显示 200  
    y = 700;  
    cout << diff() << endl;  
    // 显示 200  
}
```

请根据原理解
释看到的现象





12.5 Lambda表达式

[捕获列表](形参列表)mutable 异常说明->返回类型{函数体}

```
void f2() {  
    int x = 1000, y = 800;  
    auto diff = [&]()->int {return x - y;};  
    y = 900; // &捕获函数所有参数和函数内的局部自动变量  
    cout << diff() << endl; // 显示 100 : 1000 -900  
    y = 700;  
    cout << diff() << endl; // 显示 300 : 1000-700  
}
```

请根据原理解
释看到的现象





12.5 Lambda表达式

[捕获列表](形参列表)mutable 异常说明->返回类型{函数体}

```
auto diff = [&]( )->int {return x - y;;}
```

```
class lambda_yyyy {  
private:
```

```
    int &x;
```

```
    int &y;
```

```
public:
```

```
    lambda_yyyy(int &_x, int &_y) :x(_x), y(_y) { }
```

```
    int operator( )( ) int  
    {  
        return x - y; }
```

```
};
```

```
Lambda_yyyy diff(x, y);
```

```
cout << diff.operator( )( );
```





12.5 Lambda表达式

[捕获列表](形参列表)mutable 异常说明->返回类型{函数体}

```
void f3() {
```

```
    int x = 1000, y = 800;
```

```
    auto diff = [ &x, y]()->int {x += 200; return x - y;};
```

```
    y = 900;           // 引用Lambda表达式的外部变量
```

```
    cout << diff() << endl;    // 400 :   1200 -800
```

```
    cout << x << endl;        // 1200
```

```
    y = 700;
```

```
    cout << diff() << endl;    // 600 :   1400 -800
```

```
}
```

请根据原理解
释看到的现象





12.5 Lambda表达式

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

auto f = [ ](int e)->void {cout << e << endl;};

int main( ) {
    vector<int> v = { 1,2,3,4,5 };
    for_each(v.begin( ), v.end( ), f); // 显示 1 2 3 4 5
    return 0;
}
```





回顾

地址类型转换

数据类型转换

静态转换 `static_cast`

只读转换 `const_cast`

动态转换 `dynamic_cast` 重释转换 `reinterpret_cast`

`typeid` `const type_info&`

`auto`用于类型推导

显示调用类型转换函数 `explicit`

Lambda表达式

