華中科技大學

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

華中科技大學

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： 参数传递、返回结果的传递

12.1 隐式与显式类型转换


```
class  complex { public: int r,c;  …};
class  D : public complex {……};
complex   *a ;              D    d(…);
```

- 父类对象指针指向子类对象；父类对象引用子类对象

  a = &d;

- 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 ⇔ complex(2)

- 在运算符重载中，重载类型转换函数

  operator int( ) { return r+i; }

显式类型转换

```
int  x;          double   y;        x=int(y);
char  buf[1024];                    x=*(int *)(buf+10);
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
```

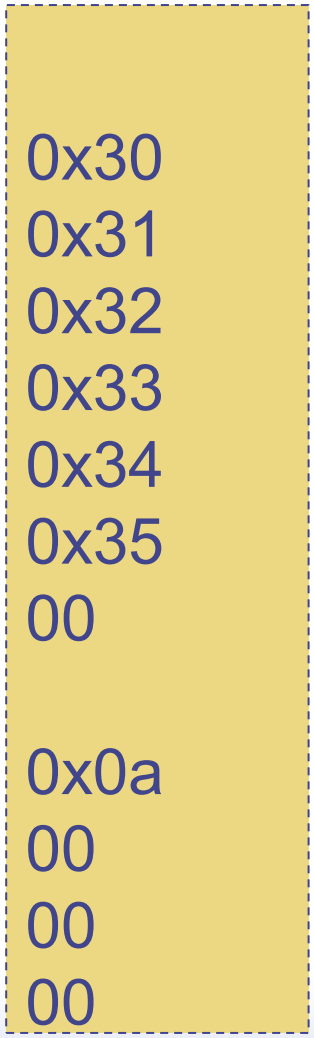
12.1 隐式与显式类型转换

華中科技大學

测验

```
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;
```



pc 0x00a3fe78

pa 0x00a3fe84

Q：运行后xxx中的值？

0x00a3fe84

0x00000030

0x33323130

xxx 0x00a3fe98

Q：pc 中的值依次是多少？

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。

12.2 cast系列类型转换

- C 语言可以进行强制类型转换；
- C 语言中太自由了，导致运行出现逻辑错误，或者一些异常；
- C++继续支持强制类型转换；
- 引入 4个强制类型转换关键字 ，对能不能转换，运行时的转换进行了更严格地检查。

12.2 cast系列类型转换

静态转换 static_cast

只读转换 const_cast

动态转换 dynamic_cast

重释转换 reinterpret_cast

……_cast<目标类型表达式>（源类型表达式）

12.2 cast系列类型转换

- static_cast 编译期检查类型能否转换；

  在转换目标为左值时，不能去除const和volitale属性；

  运行时不做动态类型检查。

- const_cast与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 基类、派生类的对象指针（或者引用）相互转换；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 将源表达式重新解释为一种新的类型。

static_cast

使用 static_cast 的场景

1.基本类型之间的转换，如从int到double，从enum到int等。但是如果转换是 narrowing（如从double到int），可能会丢失信息。

2.将指针或引用从派生类向上转换到基类（这是安全的，因为派生类对象包含基类部分）。

3.将指针或引用从基类向下转换到派生类，但是在这种情况下，使用static_cast是不安全的，因为编译器无法在运行时检查转换的正确性。通常在这种情况下，推荐使用dynamic_cast（如果基类有虚函数）。

4.任意类型与void*之间的转换，但是需要确保转换是安全的。

5.使用用户定义的类型转换函数进行转换，如果类定义了转换函数，那么static_cast可以调用这些函数。

6.将非const对象转换为const对象，这是安全的。

7.将左值转换为右值引用，例如用于实现移动语义。

static_cast

不能使用 static_cast 的场景

1.在两个不相关的指针类型之间进行转换（除非通过void中转）。例如，将int转换为double*是不允许的，因为它们是无关类型。这种情况下，需要使用reinterpret_cast。

2.在具有虚函数的基类指针和派生类指针之间进行向下转换时，如果基类没有多态性（即没有虚函数），那么static_cast可以执行向下转换，但是不安全。如果基类有虚函数（即具有多态性），则更推荐使用dynamic_cast进行向下转换，因为dynamic_cast会进行运行时类型检查。

3.转换掉const或volatile属性，static_cast不能移除const或volatile，需要使用const_cast。

4.在不同类型的成员函数指针之间或不同类型的成员指针之间进行转换，static_cast不能直接转换，可能需要reinterpret_cast。

5.在函数指针之间进行转换，如果函数类型不匹配，static_cast不能使用，需要reinterpret_cast。

static_cast

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	5.2000000000000002

y	5.1999969482421964

static_cast

- static_cast 用于在编译时进行类型转换；
- 它比C风格的转换更安全，因为它会进行一些类型检查。
- 但是它并不保证运行时的安全性，特别是在向下转换时。
- 在使用static_cast时，程序员需要确保转换是安全的。
- 对于涉及多态类型的向下转换，应优先使用dynamic_cast。
- 对于不相关的指针类型转换、移除const属性等，应使用相应的reinterpret_cast或const_cast。

static_cast 是 C++ 中最常用的类型转换运算符，但**不是万能的**。理解它的适用场景和限制对于编写安全的 C++ 代码至关重要。

const_cast

const_cast<目标类型>(源类型)

- 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

const_cast


```
const int ppp = 0;

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


*static_cast<int*>(&ppp) = 20;
                //无法从const int * 转为 int *
static_cast<int&>(ppp) = 20;
                //无法从const int转为 int &
```

dynamic_cast

dynamic_cast<T> (expr)

- 类型T是类指针、类的引用 或者 void*类型;
- expr的源类型必须是类对象的指针或引用;
  - （1）子类指针向父类指针转换
  - （2）有虚函数的基类指针向派生类指针转换
- 转换时不能去除源类型中的const和volitale属性;
- 有址引用和无址引用之间不能相互转换;

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）

讨论：自上（基类）向下（派生类）转换，为什么不安全？

dynamic_cast


```
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 {
    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
};
```

dynamic_cast

```
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。

dynamic_cast


```
B  a(3);          B  &b=a;     B  *pb;
D  c(5,7);        D &d=c;
```

```
D *pc3 = dynamic_cast<D*>(&a);    //  pc3为nullptr
pc3->f( );        // 运行时异常
```

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; 也可以执向派生类，如 pb =&c;

reinterpret_cast

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；

  足够大：能够存储一个地址或者指针；
- 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。

```
x= reinterpret_cast <int> (a);     // a 是一个类的对象
x = int(a);      //  等同 reinterpret_cast
```

cast 比较

转换类型	运算符	使用场景
静态转换	`static_cast`	编译时已知的安全类型转换
动态转换	`dynamic_cast`	多态类型的运行时安全向下转换
常量转换	`const_cast`	添加/移除 const、volatile 限定符
重新解释	`reinterpret_cast`	低级别的位模式重新解释

12.3 类型转换实例

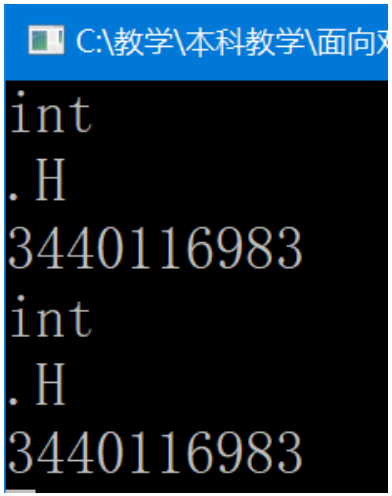
#include <typeinfo>

- 关键字typeid可以获得对象的真实类型标识
- 格式：typeid(类型表达式)；　　typeid(数值表达式);
- 返回： const type_info &
- 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;
- type_info 是一个类;
- 含有成员 const char * name( )
- 有==、!=、before、raw_name、hash_code等函数。

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;
```



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 {

    int n;
    D(int x, int y): B(x), n(y) { }
    void f( ) { cout << 'D' << endl; }
    void g( ) { cout << 'G' << endl; }
};
```

B* pb;
D* pd;

Q: 能否 pd = (D *)pb?

有条件： pb指向的就是D类对象。

12.3 类型转换实例


```
B   b(3);                    B* pb = &b;
D   d(5, 7);
D* pd(nullptr);

if (typeid(*pb) == typeid(D)) {
                             // 判断父类指针是否指向子类对象
      pd = (D*)pb;           // C语言的强制转换
      pd = static_cast<D*>(pb);
      pd = dynamic_cast<D*>(pb);    // B须有虚函数
      pd = reinterpret_cast<D*>(pb);
      pd->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 = &d; 结果如何？

12.3 类型转换实例


```
B   b(3);                    B* pb = &b;
D   d(5, 7);
D* pd(nullptr);
cout << typeid(pd).name() << endl;    //输出struct D*
cout << typeid(*pd).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(*pd).raw_name() << endl;
cout << typeid(*pd).hash_code() << endl;
```

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 x=10;              //  int  x=10;
auto p = "abcd";        //  const 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;            // tb[0] = 12   =>  ta[0]=12;
*(tb + 1) = 15;    // tb[1] = 15   =>  ta[1]=15;
```

12.4 自动类型推导


```
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;          //   *(tb2[0])=12;     tb2[0][0]=12
                         // [0] 来源于指针的编译  *p = p[0]
                         //    ta2[0[[0] = 12;
  *tb2[1] = 15;          //   *(tb2[1]) =15 =》  tb2[1][0]=15;
                         //  ta2[1][0] = 15;
  *(tb2+1)[0]=15 ;       //  *((tb2+1)[0])  => ((tb2+1)[0])[0] => tb2[1][0]
   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;
```

总结

显式类型转换　　explicit

由单参数的构造函数进行类型转换

在运算符重载中，重载类型转换函数

cast系列类型转换

static_cast、const_cast、dynamic_cast、reinterpret_cast

自动类型推导　auto