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，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。
- 赋值或函数调用时实参与形参的类型相容，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（X86编译模式）。

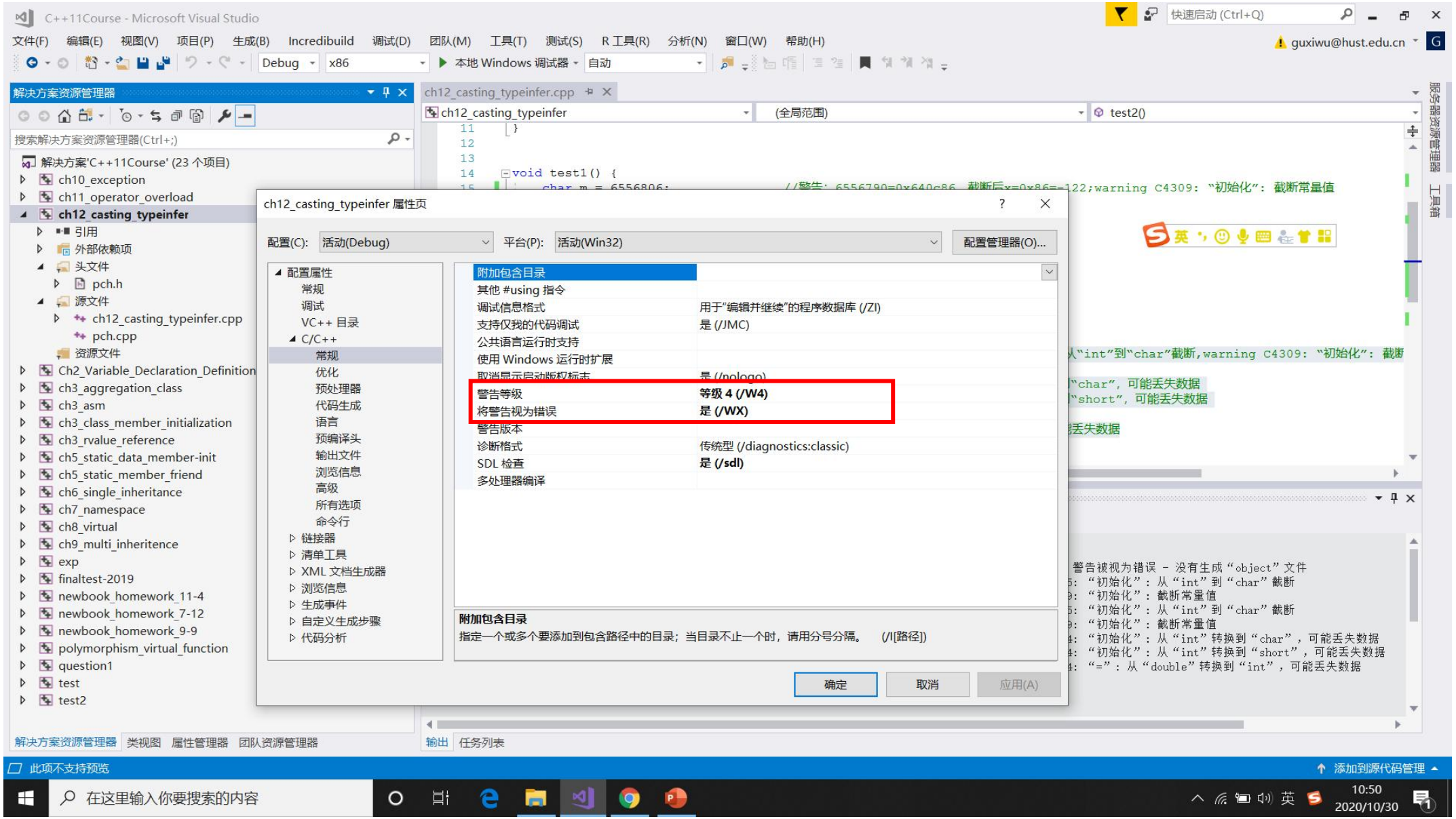
```
#include <iostream>
using namespace std;
double area(double r)
{
   return 3.14159*r*r;          //注意浮点常量3.14159，默认其为double类型
}
void main( )
{
   char m = 6556806; //警告：6556790=0x640c86，截断后x=0x86=-122;warning C4309: “初始化”：截断常量值
   int x = 2;                                  //常量2被编译程序默认当作int类型
  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(UB问题)。

```
char u = 'a'; //编译时可计算，无截断，不报警。编译时对字面量可以确定是否截断，如何截断
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，警告:“初始化”:从“int”到“char”截断，“初始化”：截断常量值
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，warning C4244: “初始化”：从“int”转换到“char”，可能丢失数据
short z = x;       //编译时不可计算，warning C4244: “初始化”：从“int”转换到“short”，可能丢失数据
double d = 3.14;
x = d;             //warning C4244 : “=”：从“double”转换到“int”，可能丢失数据
```


C++11Course - Microsoft Visual Studio
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) Incredibuild 调试(D) 团队(M) 工具(T) 测试(S) R 工具(R) 分析(N) 窗口(W) 帮助(H)
Debug x86 本地 Windows 调试器 自动
解决方案资源管理器
解决方案'C++11Course' (23 个项目)
ch10_exception
ch11_operator_overload
ch12_casting_typeinfer
引用
外部依赖项
头文件
pch.h
源文件
ch12_casting_typeinfer.cpp
pch.cpp
资源文件
Ch2_Variable_Declaration_Definition
ch3_aggregation_class
ch3_asm
ch3_class_member_initialization
ch3_rvalue_reference
ch5_static_data_member-init
ch5_static_member_friend
ch6_single_inheritance
ch7_namespace
ch8_virtual
ch9_multi_inheritence
exp
finaltest-2019
newbook_homework_11-4
newbook_homework_7-12
newbook_homework_9-9
polymorphism_virtual_function
question1
test
test2
ch12_casting_typeinfer.cpp
ch12_casting_typeinfer (全局范围) test2()
void test1() {
ch12_casting_typeinfer 属性页
配置(C): 活动(Debug) 平台(P): 活动(Win32) 配置管理器(O)...
配置属性
常规
调试
VC++ 目录
C/C++
常规
优化
预处理器
代码生成
语言
预编译头
输出文件
浏览信息
高级
所有选项
命令行
链接器
清单工具
XML 文档生成器
浏览信息
生成事件
自定义生成步骤
代码分析
附加包含目录
其他 #using 指令
调试信息格式 用于"编辑并继续"的程序数据库 (/ZI)
支持仅我的代码调试 是 (/JMC)
公共语言运行时支持
使用 Windows 运行时扩展
取消显示启动版权标志 是 (/nologo)
警告等级 等级 4 (/W4)
将警告视为错误 是 (/WX)
警告版本
诊断格式 传统型 (/diagnostics:classic)
SDL 检查 是 (/sdl)
多处理器编译
附加包含目录
指定一个或多个要添加到包含路径中的目录；当目录不止一个时，请用分号分隔。 (/I[路径])
确定 取消 应用(A)
警告被视为错误 - 没有生成"object"文件
"初始化"：从"int"到"char"截断
"初始化"：截断常量值
"初始化"：从"int"到"char"截断
"初始化"：截断常量值
"初始化"：从"int"转换到"char"，可能丢失数据
"初始化"：从"int"转换到"short"，可能丢失数据
"="：从"double"转换到"int"，可能丢失数据
输出 任务列表
此项不支持预览
添加到源代码管理
在这里输入你要搜索的内容
10:50
2020/10/30

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，编译器做了优化
   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
```

只读变量x，y，内存分配在受保护的区域。因此其值不变

```
const int z = 100; //即只读全局变量
struct A {
        const int i = 10;
}a;

void test3() {
        int x = 0;
        ((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，如是全局变量如此赋值可引起程序异常（内存页面保护）
        cout << y << endl; //还是输出9
//      ((int &)z) = 20; //试图这样修改，运行时异常。 等价于*((int *)&z) = 20;

        ((int &)a.i) = 20;          //修改为20，包含在类里的const变量不会优化，可修改
        cout << a.i << endl;    //输出20
}
```

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同时提供保护。 【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.y = 1
    x = m.q();              // x = m.y = 2;

  //int y = n.q(); //编译报错： error C2662: “int T::q(void)”：不能将“this”指针从“const T”转换为“T &”}
```

第12章 类型解析、转换与推导

◆12.2 cast系列类型转换（命名强制类型转换）

- static_cast同C语言的强制类型转换用法基本相同，但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，但能从源类型中去除const和volitale属性。
- dynamic_cast主要用于有继承关系的基类型和派生类型之间的相互转换。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通常为运算对象的位模式提供低层级的重新解释。主要用于名字（运算对象）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（能够存放地址）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T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)。
- 只要不转换底层const，都可以使用static_cast（后面补充顶层和底层const）

```
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 test_static_cast() {
    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
    //*static_cast<int*>(&x) = 0; //错误：无法去除const将源类型const int *转换为int *. 不等价于*（int *)&x=0;
    //上面的例子也说明了static_cast和传统强制类型转换的区别
    //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
}
```

【例12.6】使用static_cast对数值表达式进行强制类型转换。

顶层和底层const

- 顶层const：表示任意的变量是常量（最靠近变量名的const）（引用除外）
- 底层const：和指针与引用等复合类型的基本类型有关。修饰基本类型的const

```
int i = 0;
int * const p1 = &i;          //顶层const，修饰p1，最靠近变量p1
const int ci =42;             //顶层const，修饰变量c1
const int *p2 = 0;            //底层const，修饰指针所指向对象的类型int（基本类型）
const int * const p3 = 0; //靠右的const的顶层const，靠左的const是底层const

const int &r = i; //修饰引用的const都是底层const。（因为引用相当于*const p，const指针）
```

```
//顶层和底层const

//static_cast：只要不转换底层const，都可以用
const char *p1 = 0;                    //底层const
//char *q1 = static_cast<char *>(p1);     //错误。只有不转换底层const，static_cast才可用
const int &r = 1; //底层const
//int &rr = static_cast<int &>(r);          //错误。只有不转换底层const，static_cast才可用
const int c1 = 0; //顶层const
int c2 = static_cast<int>(c1);             //只要不转换底层const，static_cast可用
const char *const p2 = 0;
const char * p3 = static_cast<const char *>(p2); //只转换了顶层const，正确
//const char * p4 = static_cast<char *>(p2); //错误，转换了底层const
char *const p5 = 0; //顶层const
char *p6 = static_cast<char *>(p5);                //OK，只要不转换底层const，static_cast可用
```

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<目标类型表达式>(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(运行时检查是否真的可以转换为目标类型)。
- const_cast的目标类型必须是指针或引用或指向对象类型成员的指针，并且去掉const或volitile属性，但不能改变指针或引用的基础类型
- const_cast不能将无址常量、位段访问、无址返回值转换为有址引用(和C的强制转换相同)

```
//const_cast的目标类型必须是指针或引用或指向对象类型成员的指针
const int c3 = 0;
//int c4 = const_cast<int>(c3);       //错误。const_cast类型必须是指针或引用或指向对象类型成员的指针
char c4 = 0;
//int c5 = const_cast<int>(c4);       //错误。const_cast类型必须是指针或引用或指向对象类型成员的指针
int &rc = const_cast<int &>(c3);     //正确。目标类型是int &

const char * const p7 = 0;
char *p8 = const_cast<char *>(p7); //正确，去掉了指针的顶层和底层const
const char *p9 = 0;
char *p10 = const_cast<char *>(p9);          //正确。去掉了指针的底层const
char *p11 = 0;
char *p12 = const_cast<char *>(p11); //正确，没有意义。

int c7 = 0;
const int &r1 = c7; //引用的const是底层const
int &r2 = const_cast<int &>(r1); //正确，去掉了底层const

//const_cast只能去掉const性质（cast away const），但不能改变类型，这时和static_cast的一个重要区别
const char * p14 = 0;
//int *p15 = const_cast<int *>(p14); //错误。const_cast只能调节类型限定符，不能更改基础类型
```

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
}
```

```
void main( ) {
   Test o(7);                                  //o.number=7，o.nn=7
   o.dec();                                    //o.number=6，o.nn=6,Test *可以传递给const TEST* const this
   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
   o.*const_cast<int Test::*>(&Test::nn) = 3;  //对象实例成员不受保护可修改：o.nn=3，将const int TEST::*p转int TEST::*p
   ww = o.nn;                                  //ww=3
   const_cast<volatile int&>(yy) = 4;//添加volatile，引用yy无内存不受保护，yy=4.const int &转volatile int &。static属性不变
   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。非const转const正确
   //ww = static_cast<const int&>(zz);         //错误，static_cast不能用于底层const
   ww = static_cast<const int>(zz);            //正确，只要不包含底层const，static_cast都可以用
   ww = *const_cast<const int*>(&zz);          //正确：ww=6。非const转const正确
   const_cast<volatile int&>(ww) = 5;          //正确：添加volatile：ww=5
  //非const转const用static_cast和const_cast都可用（volatile类似）。但要注意const_cast的目标类型必须是指针、引用
  //static_cast不能用于底层const
  int x = 0;  int *p1 = &x;
  const int i2 = static_cast<const int>(x);
  const int *p2 = static_cast<const int *>(p1);      ////p1无const
  const int *p3 = const_cast<const int *>(p1);
  int *const p4 = static_cast< int *const>(p1);
  int *const p5 = const_cast<int *const>(p1);}
```

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（包括常量对象或变量对象：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。
- 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。转换失败得到空指（指针类型的dynamic_cast）针或出现异常（引用类型的dynamic_cast)

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
} b;
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 g(){ cout << n << endl; }
};
void main( ) {
    B a(3);
    B &b=a;
    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
pc1->g();                         //输出随机数，因为a不是D类型对象，因此static_cast<D*>(&a);是不安全的
D *pc2 = static_cast<D*>(&b);     //语法正确但为不安全的自上向下转换.b是a的引用，和上面等价
pc2->f();                         //输出B

D *pc3 = dynamic_cast<D*>(&a);//由于a不是D类型对象，转换返回空指针。可看出dynamic_cast的好处
pc3->f();                         //运行异常：pc3为nullptr（a非子类D的对象）这里可看出dynamic_cast的好处，
可以运行时进行类型判断，更安全

D *pc4 = dynamic_cast<D*>(&b);//b是a的引用，由于a不是D类型对象，转换返回空指针
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。c就是D类型对象。转换后的D*赋值给B*
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，d是c的引用。转换后的D*赋值给B*
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
```

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

```
D &ra2 = static_cast<D&>(b);     //语法正确但不为安全的自上向下转换,b不是D类型对象
ra2.f();                         //输出B：根据虚函数入口地址表首址

D &rra2 = dynamic_cast<D&>(b); //b不是D类型对象，会抛出运行时异常。可看到dynamic_cast的好处

B &rc1 = dynamic_cast<D&>(c);  //语法正确且为安全的自下向上赋值,c就是D类型对象
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，d引用了D类对象
rc2.f();                         //输出D：正确的多态行为

//把左值转换成右值引用的另外一种方法。（不用std::move)
B &&rc3 = static_cast<D&&>(c);  //语法正确且为安全的自下向上赋值，c就是D类对象。
rc3.f();                         //输出D：正确的多态行为
//把左值转换成右值引用的另外一种方法。（不用std::move)
B &&rc4 = dynamic_cast<D&&>(c); //语法正确且为安全的自下向上赋值，c就是D类对象
rc4.f();                         //输出D：正确的多态行为（gcc下）。 VS2019抛出运行时异常

B &&rc5 = static_cast<D&&>(d);  //语法正确且为安全的自下向上赋值,d就是D类对象
rc5.f();                         //输出D：正确的多态行为（gcc下）。

B &rc6 = dynamic_cast<D&>(rc5); //正确：自上向下转换，自下向上赋值，rc5引用的就是D类对象，B&&，左值
rc6.f();                          //输出D：正确的多态行为
}
```

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

```
//指针类型的dynamic_cast
void test_dynamic_cast2() {
        Base b, *pb1 = &b;
        Derived d, *pb2 = &d;
        //pb1指向Base对象，转换为Derived失败
        if (Derived *pd = dynamic_cast<Derived *>(pb1)) {
                cout << "转换成功" << endl; pd->f();
        }
        else {//若pb1指向的对象不是Derived类型，转换失败，返回nullptr
                cout << (pd == nullptr) << endl; cout << "转换失败, pd = nullptr" << endl;
        }
        //pb2指向Derived对象，转换为Derived成功
        if (Derived *pd = dynamic_cast<Derived *>(pb2)) {
                cout << "转换成功" << endl; pd->f();
        }
        else {//若pb2指向的对象不是Derived类型，转换失败，返回nullptr
                cout << (pd == nullptr) << endl; cout << "转换失败, pd = nullptr" << endl;
        }
}
```

```
class Base {
public: virtual void f() { cout << "Base:f()" << endl; }
};
class Derived :public Base {
public: virtual void f() { cout << "Devived:f()" << endl; }
};
```

在if条件语句里，执行dynamic_cast操作同时检查转换返回的指针是否为nullptr，可以确保类型转换和结果检查在同一条语句完成。对空指针执行dynamic_cast，返回所需类型的空指针

```
//引用类型的dynamic_cast
//和指针转换不同，因为不存在空引用，因此：如果b引用的不是Derived对象，则会抛出异常
void test_dynamic_cast_ref(const Base &b) {
        try {
                const Derived &d = dynamic_cast<const Derived &>(b);
        }
        catch(std::bad_cast){//该异常类型定义在typeinfo标准库头文件
                //处理转换失败的异常
        }
}
```

和指针转换不同，因为不存在空引用，因此：如果b引用的不是Derived对象，则会抛出异常

```
class A {
//        virtual void f() {};
};

class C:public A{
public:
        C( ) = default;
};

void test() {
        A *p3 = new C;

        //编译错误. 运行时dynamic_cast的操作数p3必须是多态类型。
        //因为A没有虚函数，也就没有虚函数表，因此A不是多态类型.
        //若在A里加上虚函数，这个语句就可以
        C *p4 = dynamic_cast<C *>(p3); // 基类向派生类转换，基类必须具有多态性

}
```

dynamic_cast使用注意事项

第12章 类型解析、转换与推导

◆reinterpret_cast——重释转换

- 关键字reinterpret_cast实现有址表达式（有名字）到指针或(有址或无址)引用类型的转换以及指针与足够大整数类型间的相互转换。
- 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。
- 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
```

```
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

//整数和指针相互转换
//int f = reinterpret_cast <int> (e);                  //指针e转为整型，赋给f。gcc编译报错。因为Linux 64位系统上指针类型占用8个字节，而int类型占用4个字节，所以会出现loses precision. VS下指针可以转int
long long f = reinterpret_cast <long long> (e);
printf("e=%p, f=%x\n", e, f);                          //打印指针e和f，f以16进制显示，e==f
B *g = reinterpret_cast <B *> (f);                     //整型f转为B *，赋值给g=&a
printf("e=%p, f=%x\n", g, f);                          //打印指针e和f，f以16进制显示，e==f

//名字转引用（左值引用和右值引用）
B &h = reinterpret_cast <B&> (a);                      //有址变量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=b.m=a.m=3

//类的静态数据成员
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;
```

第12章 类型解析、转换与推导

```
//类的静态函数
   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
   void(&m)() = reinterpret_cast<void(&)()>(B::e);       //名字B::e转函数引用，void(&m)()=B::e
   m();                                                  //等价于调用B::e()，输出E

   //类的实例成员函数
   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

   //类的实例成员
   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（编译时）。有时需要调用子类定义的新函数（父类没有该函数，这是需要父类型转子类型）
- 武断或盲目地向下转换（父类型转子类型），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; }                  //子类新定义的函数
};
```

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;                                //父类引用指向父类对象
    D c(5, 7);                               //定义子类对象c
    D &d = c;                                //子类引用指向子类对象
    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;                   //父类指针指向子类对象
    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，当if条件满足时，转换是安全的
        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，编译时pb无法调用g( )
    }
    cout << typeid(pc).name( )<<endl;        //输出struct D*，gcc下输出P1D
    cout << typeid(*pc).name( ) << endl;     //输出struct D。gcc下输出1D
    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**。auto让编译器通过初始值来推算类型，因此auto定义的变量**必须有初始值**；
- 可用于推导变量类型、各种函数的返回值类型、以及类的有const或inline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，inline。
- 使用auto可以在一条语句中声明多个变量，但必须保证这一条声明语句只能有一个基本数据类型

```
auto i =0, *pi = &i;          //ok, i是整型，p是整型指针。基本数据类型一致
auto sz = 0,  pi = 3.14;  //错误：sz和pi的字面量类型不一致
```

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明确说明了变量类型int
static auto x = 3;                    //正确：推导定义“static int x = 3;”
class A {
   auto const static m=3;             //可以对类的const静态成员推导
   inline auto const volatile static n=x; //使用inline时可使用任意表达式初始化
public:
   // auto i = 0; //不能对类的非静态成员自动推断，因为非静态成员必须有对象才存在，编译时没对象
   inline static auto c = 10; // //可以对类的inline静态成员推导
};
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：
  - “数组名[表达式][表达式]”表示除第一、二维外的数组类型
  - int a[10][10]; //a[1]的类型为int [20]
- 被推导类型取决于初始化表达式，如果初始化表达式为数组类型的第1维（仅用数组名）或者剩下维（使用“数组名[表达式]”）的第1维，则优先被解释为指针。
- 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)，为函数
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]。a[10][20]被解释为指针时去掉第1维，
为int (*)[20], a[1]的类型为int [20],如果被解释为指针，去掉第1维([20])，解释为int *
auto  *h = a[1];                          //选择int *h而非int h[20]，*h明确告诉编译器按指针解释
int arr[10];
auto arr2 = arr;
printf("%s\n", typeid(arr2).name());      //输出int *
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)，*明确告诉编译器按指针解释
```

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

```
void test_auto3() {
        //如果初始化表达式是引用，则去除引用语义:按被引用对象的类型推断。
        int a = 10; int &b = a;
        auto c = b;//c的类型为int而非int&（去除引用）
        auto &d = b;//此时c的类型才为int&

        //auto会忽略顶层const，保留底层const
        int i = 0, &r = i;
        const int ci = i, &cr = ci;

        auto b = ci; //b是int，忽略顶层const
        auto c = cr; //c是int，首先去除引用语义，按被引用ci推导，忽略ci的顶层const
        auto d = &i; //d是int *
        auto e = &ci; //e是const int *，&ci是const int *,保留底层const
        //如果希望推导出顶层const，需要明确指出
        const auto f = ci; //f是const int

        //如果auto关键字带上&号，则不去除const语意
        const int a2 = 10;
        auto &b2 = a2;//因为auto带上&，故不去除const，b2类型为const int
        //因为如果去掉了const，则b2为a2的非const引用，通过b2可以改变a2的值，则显然是不合理的。
}
```

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。
  - decltype(f()) sum = x;
  - 用函数f的返回类型来声明变量sum，并初始化为x
  - 发生在编译时，不会调用函数f
- 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];”
//函数返回类型是int (*)[20], 参数是int [10][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int [10][20]
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);
}
```

```
//decltype处理顶层const、引用和auto不一样。decltype会返回变量的完整类型（包括顶层const和引用）
const int ci = 0, &cj = ci;
decltype(ci) x = 0;                    //x的类型是const int
//x = 2;                               //错误，x是const int
auto ax = ci; ax = 2;                  //ax类型是int
decltype(cj) y = x;                    //y类型是const int &
//decltype(cj) z;                      //错误，/cj类型是const int &，引用必须初始化
auto z = cj; z = 2;                    //z的类型是int

//顾名思义，delctype返回的是变量的声明类型，所以保留引用、const特性
//const int i = 0; //i是实际类型是int，但不能修改（const是类型修饰符），声明类型是const int

//引用从来都是作为被引用对象的同义词出现，decltype是例外
//decltype(表达式)返回的是表达式的类型。如果表达式返回左值，则decltype得到是引用类型
int i = 42, *p = &i, &r = i;
decltype(++i) x = i; //x是int &，必须初始化,如果去掉=i，就会报错
decltype(*p) y = i;  //指针解引用操作*p是左值表达式，因此y是int &，必须初始化
decltype(r + 0) z;   //r + 0是右值，因此z是int

//decltype((变量))的结果永远是引用。decltype(变量)的结果只有当变量是引用时，结果才是引用
decltype(i) j; //j是int，没有初始化
decltype((i)) k = r; //k是int &，必须初始化
//原因：(i)被解释为作为赋值语句左值的特殊表达式
```

第12章 类型解析、转换与推导

- Lambda表达式是C++引入的一种匿名函数。存储Lambda表达式的变量被编译为临时类的对象。
- 该临时类变量的对象被构造时，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operator( ) (…)；当调用f(3)时，等价于f.operator( )(3)
  - Lambda表达式被编译成函数对象，函数对象名为f

重载函数调用操作符（ ）

为多目运算符

```
int sum(int x, int y) { return x + y;}
int s = sum (1,2); //三个操作数sum, 1, 2
//第一个操作数为函数名，这里把函数名看做函数对象
```

()只能通过类的普通成员函数重载，这意味着()第一个操作数必须是this指针指向的类对象，这样的对象称为函数对象。

```
class AbsInt{
public:
   //语法：returnType operator( ) (参数列表)
   //调用：objectofAbsInt(实参)，类AbsInt的对象称为函数对象
   int operator()( int val) { return val > 0? val:-val; }
} absInt;
int i = absInt(-1); //函数对象可以作为函数的参数，这意味着我们可以将函数当做对象传递. 在模板编程里广泛使用。在C+11里，还可以用lamda表达式
```

函数指针也可以作为函数的参数，但函数指针主要的缺点是：
被函数指针指向的函数是无法内联的。而函数对象则没这个问题。

```
void  main() {
   int m = 1, p = 3;
   const int n = 2, q = 4;
   auto  g = [m, n, &p, &q](int x) ->int { p++; /*错： m++; n++; q++;*/; return m+n+x; };
   int z = g(0);                  //m=1,p=4,z=3, 等价于g.operator()(0)
   z = g(0);          //m=1,p=5,z=3
}
//为匿名类对象g生成的匿名类及其函数如下
class 匿名类{
    const int  m, n;          //Lambda表达式无mutable时， 非&捕获的可写外部变量m为const成员
    int &p;                   //&捕获的外部变量保持其原有类型int+&
    const int &q;             //&捕获的外部变量保持其原有类型const int+&
public:
    构造函数(int m, const int n, int &p, const int &q) : m(m), n(n), p(p), q(q){}
    int operator()(int x) { p++; return m + n + x; }
}g(m, n, p, q);    //调用构造函数初始化匿名类对象g
```

m, n为值捕获;p，q为引用捕获

当捕获列表以值的方式传递后，lambda表达式不能修改这个变量的值，只能使用

引用捕获的非const值可以修改

```
void  main() {
   int m = 1, p = 3;
   const int n = 2, q = 4;
   auto  f = [m, n, &p, &q](int x)mutable->int { p++; /*错:n++;q++;*/ return m++ + x; };
   int z = f(0);                  //m=1,p=4, z=1, 等价于f.operator()(0);
   z = f(0);          //m=1,p=5, z=2 （匿名类里的m变成2）
}
//为匿名类对象f生成的匿名类及其函数如下
class 匿名类{
    mutable int  m;                        //Lambda表达式有mutable时， 非&捕获的可写外部变量m为
mutable成员
    const int n;                //保持n的原有类型为const int
    int &p;                     //保持p的原有类型为int+&
    const int &q;               //保持q的原有类型为const int+&
public:
    构造函数(int m, const int n, int &p, const int&q):m(m), n(n), p(p),q(q){ }
     int operator()(int x) {p++; /*错:n++;q++;*/ return m++ + x; }
}f(m, n, p, q);                 //调用构造函数初始化匿名类对象f
```

m, n为值捕获;p，q为引用捕获

，lambda表达式有mutable时，值捕获的非const值变成mutable，即可以修改

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”表示引用捕获外部变量，[&]表示引用捕获所有函数参数或函数定义的局部自动变量。其可写特性同被捕获的外部变量一致。
- 出现“=变量名”表示值捕获外部变量的值（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表达式中可以修改“值参传递的可写变量的值”，但调用后不影响Lambda表达式捕获的对应该可写外部变量的值（值捕获）。

```
//例12.21
int main( ) {                                                    准函数：函数对象不包含其他成员，除了operator()(...)
   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，可以用函数指针h指向“准函数”
   h = f;                                                        //正确：f的Lambda表达式捕获列表为空，f当准函数使用
   auto m = [a, f](int x)->int { return x * f(x); };             //m是函数对象：捕获a初始化实例成员
   //int(*k)(int)=[a](int x)->int{return x;};      //函数指针k不能指向函数对象（捕获列表非空,函数对象包含其他成员）
   //h = m;                                        //错误：m的Lambda表达式捕获列表非空，m是函数对象
   //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,f是函数对象，显式调用operator()()
   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
}
```

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”。
- 早期编译器实例函数成员中的Lambda表达式默认捕获this，故它是一个准对象。但vs2019必须明确捕获this才能访问实例数据成员。
- 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,b值捕获，其它引用捕获
         a++;                       //f()的参数a被值捕获并传给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，创建函数对象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，a是引用捕获
        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，b是值捕获
        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; }; //静态成员函数g没有this，没有this被捕获，创建对象k
    auto p = k;                              //p的类型为class<lambda…>
    int (*q)(int) = k;                       //问题：若将红色部分放入void f(int &a)中，如何？ int (*q)(int) = k;就不成
立
```

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
}
```