**C++ typeid关键字详解**

**typeid关键字**

注意：**typeid是操作符，不是函数**。这点与sizeof类似）

运行时获知变量类型名称，可以使用 typeid(变量).name()

需要注意不是所有[编译器](https://so.csdn.net/so/search?q=%E7%BC%96%E8%AF%91%E5%99%A8&spm=1001.2101.3001.7020)都输出”int”、”float”等之类的名称，对于这类的编译器可以这样使用

int ia = 3;

if(typeid(ia) == typeid(int))

{

cout <<"int" <<endl;

}

* 1
* 2
* 3
* 4
* 5

**RTTI（Run-Time Type Identification)-运行时类型识别**

在揭开typeid神秘面纱之前，我们先来了解一下RTTI（Run-Time Type Identification，运行时类型识别），它使程序能够获取由基指针或引用所指向的对象的实际派生类型，即允许“用指向基类的指针或引用来操作对象”的程序能够获取到“这些指针或引用所指对象”的实际派生类型。

在C++中，为了支持RTTI提供了两个操作符：dynamic\_cast和typeid

* dynamic\_cast允许运行时刻进行类型转换，从而使程序能够在一个类层次结构中安全地转化类型，与之相对应的还有一个非安全的转换操作符static\_cast，因为这不是本文的讨论重点，所以这里不再详述，感兴趣的可以自行查阅资料。
* typeid是C++的关键字之一，等同于sizeof这类的操作符。typeid操作符的返回结果是名为type\_info的标准库类型的对象的引用（在头文件typeinfo中定义，稍后我们看一下vs和gcc库里面的源码），它的表达式有下图两种形式。

**实现机制与使用技巧**

**type\_info类对象类别判别**

**对象类别判别分析**

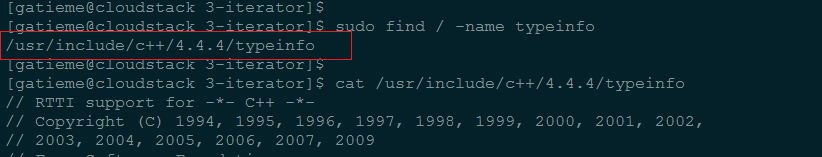
如果表达式的类型是类类型且至少包含有一个虚函数，则typeid操作符返回表达式的动态类型，需要在运行时计算；  
否则，typeid操作符返回表达式的静态类型，在编译时就可以计算。

ISO C++标准并没有确切定义type\_info，它的确切定义编译器相关的，但是标准却规定了其实现必需提供如下四种操作（在之后的章节中我会来分析type\_info类文件的源码）

| **运算** | **描述** |
| --- | --- |
| t1 == t2 | 如果两个对象t1和t2类型相同，则返回true；否则返回false |
| t1 != t2 | 如果两个对象t1和t2类型不同，则返回true；否则返回false |
| t.name() | 返回类型的C-style字符串，类型名字用系统相关的方法产生1 |
| t1.before(t2) | 返回指出t1是否出现在t2之前的bool值 |

type\_info类提供了public虚 析构函数，以使用户能够用其作为基类。它的默认构造函数和拷贝构造函数及赋值操作符都定义为private，所以不能定义或复制type\_info类型的对象。程序中创建type\_info对象的唯一方法是使用typeid操作符（由此可见，如果把typeid看作函数的话，其应该是type\_info的 友元）。type\_info的name成员函数返回C-style的字符串，用来表示相应的类型名，但务必注意这个返回的类型名与程序中使用的相应类型名并不一定一致（往往如此，见后面的程序），这具体由编译器的实现所决定的，标准只要求实现为每个类型返回唯一的字符串。

**type\_info类源代码**

使用sudo find / -name typeinfo.h来查找源码  


#ifndef \_TYPEINFO

#define \_TYPEINFO

#include <exception>

namespace std

{

class type\_info

{

public:

virtual ~type\_info();

{ return \_\_name[0] == '\*' ? \_\_name + 1 : \_\_name; }

bool before(const type\_info& \_\_arg) const

{ return \_\_name < \_\_arg.\_\_name; }

bool operator==(const type\_info& \_\_arg) const

{ return \_\_name == \_\_arg.\_\_name; }

bool operator!=(const type\_info& \_\_arg) const

{ return !operator==(\_\_arg); }

virtual bool \_\_is\_pointer\_p() const;

virtual bool \_\_is\_function\_p() const;

protected:

const char \*\_\_name;

explicit type\_info(const char \*\_\_n): \_\_name(\_\_n) { }

private:

type\_info& operator=(const type\_info&);

type\_info(const type\_info&);

};

} // extern "C++"

#endif

**示例1-基本数据类型**

下表列出了使用typeid操作符的表达式的值

int a;

double b;

char \* c;

long d;

| **运算** | **描述** |
| --- | --- |
| typeid(a) == typeid(int) | true |
| typeid(a) == typeid(float) | false |
| typeid(a) == typeid(int \*) | false |
| typeid(b) == typeid(double) | true |
| typeid(b) == typeid(float) | false |
| typeid(b) == typeid(long double) | false |
| typeid(c) == typeid(char \*) | true |
| typeid(c) == typeid(char) | false |
| typeid(c) == typeid(string) | false |
| typeid(d) == typeid(long) | true |
| typeid(d) == typeid(int) | false |

操作符typeid返回的是一个type\_info类（用于描述数据类型的一个系统类）对象的引用。这个操作符可以用于表达式和类型名（包括自定的数据类型，比如类）。

**示例2-类对象**

class base

{

public :

void m(){cout<<"base"<<endl;}

};

class derived : public base

{

public:

void m(){cout<<"derived"<<endl;}

};

假设我们根据例2中定义的两个类来定义如下指针：

base \* p = new derived;

下表将给出使用typeid操作符的结果。

| **运算** | **描述** |
| --- | --- |
| typeid(p) == typeid(base\*) | true |
| typeid(p) == typeid(derived\*) | false |
| typeid(\*p) == typeid(base) | true |
| typeid(\*p) == typeid(derived) | false |

对于表达式typeid(p)，同样，因为p是base\*类型的指针，因此typeid(p) == typeid(base\*)为真，而typeid(p) == typeid(derived\*)为假。而对于表达式typeid(\*p)，由于此时的基类不具有多态性，因而\*p将会采用编译期类型来计算，编译期\*p是base对象，因此表达式typeid(\*p) == typeid(derived)为假，typeid(\*p) == typeid(base)为真。

**示例3-带虚函数的基类**

class base

{

public :

virtual void m(){cout<<"base"<<endl;}

};

class derived : public base

{

public:

void m(){cout<<"derived"<<endl;}

};

假设我们如本例所示定义了两个类base类和derived类，基于这两个类定义，我们定义指针如下：

base \* p = new derived;

下表将给出使用typeid操作符的结果。

| **运算** | **描述** |
| --- | --- |
| typeid(p) == typeid(base\*) | true |
| typeid(p) == typeid(derived\*) | false |
| typeid(\*p) == typeid(base) | false |
| typeid(\*p) == typeid(derived) | true |

对于表达式typeid(p)，因为p是base\*类型的指针，因此typeid(p) == typeid(base\*)为真，而typeid(p) == typeid(derived\*)为假。而对于表达式typeid(\*p)，因为base类具有多态性，因而在计算typeid(\*p)时会根据运行时p所指向的实际类型去计算，而本例中p指向的是派生类对象，因此表达式typeid(\*p) == typeid(derived)为真，typeid(\*p) == typeid(base)为假。

**异常处理bad\_typeid**

class bad\_typeid : public exception

{

public:

bad\_typeid () throw() { }

// This declaration is not useless:

// http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc\_6.html#SEC118

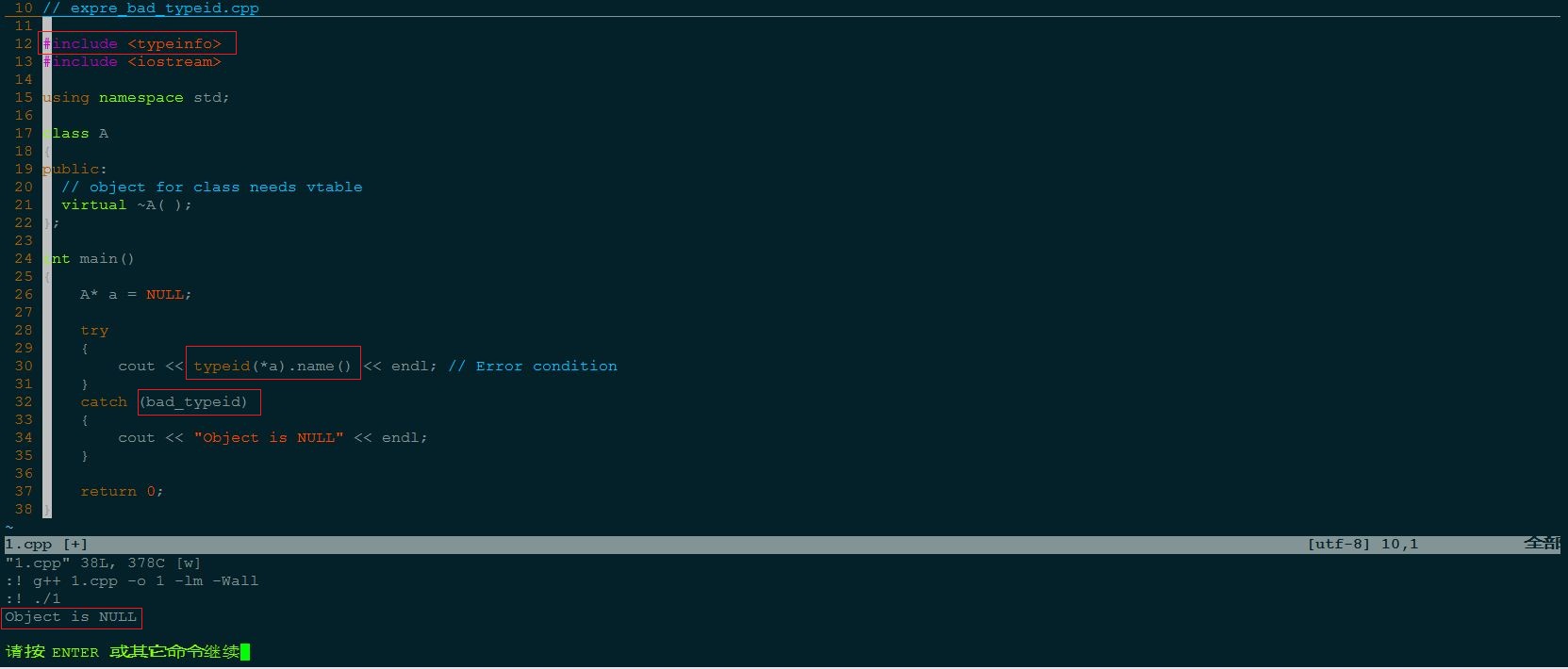
virtual ~bad\_typeid() throw();

// See comment in eh\_exception.cc.

virtual const char\* what() const throw();

};

} // namespace std



参照

[【C++】typeinfo.h](http://www.cnblogs.com/visayafan/archive/2011/11/29/2268135.html)

[C++中的typeid关键字](http://blog.csdn.net/goodlixueyong/article/details/6244303)