参考<https://bbs.pediy.com/thread-61873.htm>

注意：在windows平台下面

**背景：**

RTTI(RuntTime TypeInfo)，C++运行时类型识别

当一个类的引用或者指针的值去进行类型识别，是动态识别

当一个类的实例去进行类型识别，是静态识别

动态识别需要编译打开/GR开关才有效，否则会引起一个运行时错误。

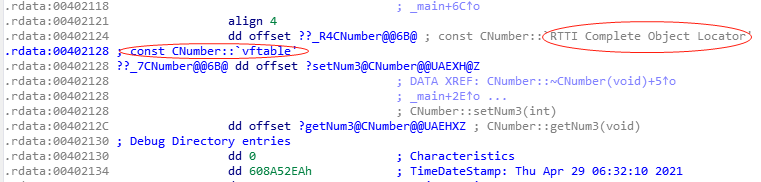
**实现原理：**

C++一个类存在虚函数，或者的基类存在虚函数，或者virtual继承了其他类，或者开启了RTTI，那么编译器会为该类生成一个虚表。

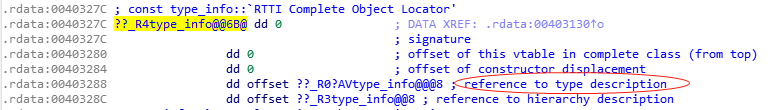
一般类的头4个字节为该表指针。（当基类是多重继承的时候，非第一个继承的基类虚表不是在头4字节）。

当有RTTI在的时候，虚表-4的位置是一个叫做RTTI complete object locator的描述体。

（windows x86下）该描述体+c的位置是一个叫做RTTI type descriptor的描述体，该描述体+8的位置是一段原始类名的buf，我们只要得到该buf，就能得到类的名字。



RTTI complete object locator的描述体+c的位置是一个叫做RTTI type descriptor的描述体



RTTI type descriptor的描述体+8的位置是一段原始类名的buf

**实现：**

RTTI type descriptor这个描述体在C++中叫做type\_info，它的定义如下：

class type\_info{

public:

\_CRTIMP virtual ~type\_info();

\_CRTIMP init operator==(const type\_info& rhs) const;

\_CRTIMP int operator!=(const type\_info& rhs)const;

\_CRTIMP int before(const type\_info%rhs)const;

\_CRTIMP const char\* name()const;

\_CRTIMP const char\* raw\_name()const;

private:

void\* \_m\_data;// *\_UndecoratedName*

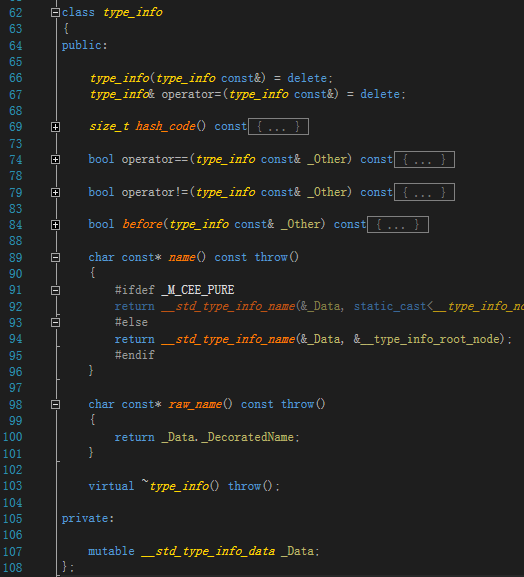
char \_m\_d\_name[1];// \_DecoratedName

type\_info(const type\_info& rhs);

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

};

查看vs2015中type\_info



struct *\_\_std\_type\_info\_data*

{

char const\* *\_UndecoratedName*;

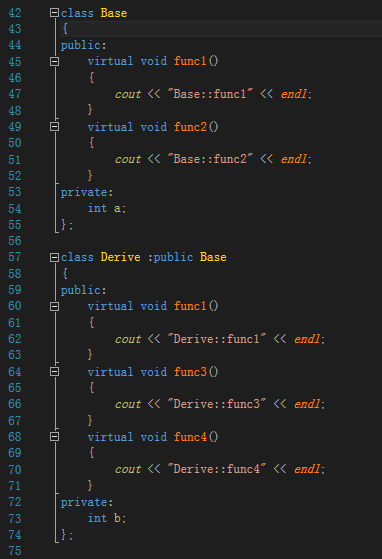
char const \_DecoratedName[1];

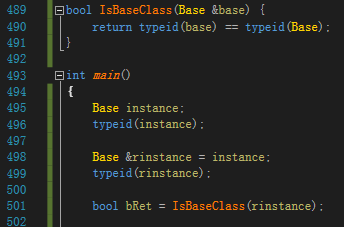
};

decorate是装饰的意思，\_DecoratedName表示编译之后的名称，*\_UndecoratedName*表示写代码时候的名称

typeid作为C++关键字，给它传递一个任意类型的实例，它能够返回一个type\_info的引用。

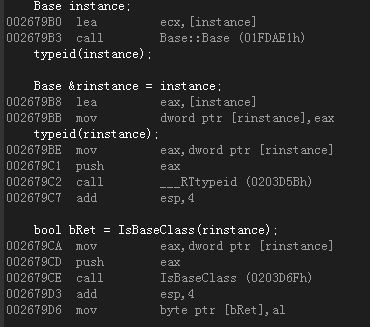
例如：





RTTI类型识别的实质就是strcmp前面说的buf(type\_info中的\_m\_d\_name)是不是和Base的\_m\_d\_name相等

C++采用静态识别，对应的汇编代码为



前面所说buf(\_m\_d\_name，vs2015中是\_Data.\_DecoratedName)其实是一个C++类名的内部表达式，比如我们的类Base的\_m\_data（vs2015中是\_Data.*\_UndecoratedName*）其实显示为?AVBase@@，怎么还原成我们认识的class Base这种形式呢。

参考下面代码

*printf*("%s", typeid(rinstance).name());

可以正确显示出class Base，所以我们如果有了raw\_name(前文的buf，即type\_info的\_m\_d\_name，或者是vs2015的\_Data.\_DecoratedName)，可以通过type\_info的const char\* name()const函数来实现。

现在怎么调用name()这个函数，可以构造一个假的type\_info\*，然后去访问他的代码如下

struct \_fake\_typeinfo {

void\* vtbl;

void\* name;

char raw\_name[1024];

\_fake\_typeinfo(const char\* rname) : vtbl(nullptr), name(nullptr) {

*strcpy*(raw\_name, rname);

}

};

\_fake\_typeinfo ti(".?AVBase@@");

*type\_info*\* pti = reinterpret\_cast<*type\_info*\*>(&ti);

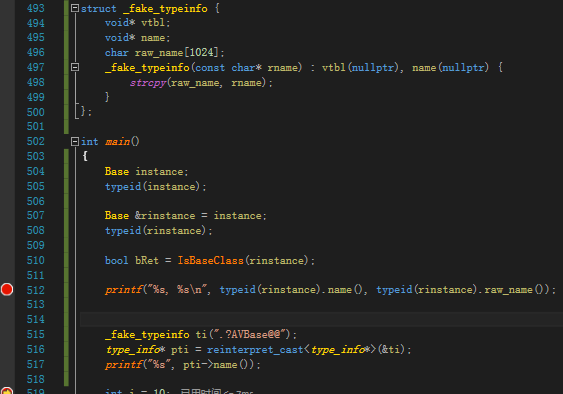
*printf*("%s", pti->*name*());

可以看到正确的显示出了class Base了

这里结构\_fake\_typeinfo是根据type\_info的定义模拟出来的（Linux平台目前看下来不一样），sizeof(type\_info) = 12

前四个字节存的是虚表指针，后面是一个*\_\_std\_type\_info\_data*类型的数据\_Data，数据是

char const\* *\_UndecoratedName*; char const \_DecoratedName[1];



IDA无法使用C++的定义结构体调用C++ type\_info的函数，但是脚本支持2个指令exec和readstr。可以认为他是idc跟外界打交道的两个途径。

当需要把一个raw\_name(\_DecoratedName)转换为一个name（*\_UndecoratedName*）的时候，idc先执行一个外部程序，把raw\_name作为commandline传递

该外部程序把name生成一个文件，然后idc读取改文件的第一行来得到返回值。

该脚本需要光标在有RTTI的工程类的虚表的-4的位置（即RTTI complete object locator所在的位置）执行。

并且存在一个c:\\111.exe来翻译name

执行后会给RTTI complete object locator备注上类的显示名字。

111.exe的源码：

#include "stdafx.h"

#include <iostream>

struct \_fake\_typeinfo {

void\* vtbl;

void\* name;

char raw\_name[1024];

\_fake\_typeinfo(const char\* rname) : vtbl(nullptr), name(nullptr) {

*strcpy*(raw\_name, rname);

}

};

int main(int argc, char\*\* argv)

{

//\_fake\_typeinfo ti1("\_ZTI7CNumber");

//type\_info \*p\_tinfo1 = reinterpret\_cast<type\_info \*>(&ti1);

//printf("%s", p\_tinfo1->name());

*FILE* \*fp = *fopen*("C:\\111.txt", "w");

if (argc != 2) { *fclose*(fp); return -1; }

\_fake\_typeinfo ti(argv[1]);

*type\_info* \*p\_tinfo = reinterpret\_cast<*type\_info* \*>(&ti);

*fprintf*(fp, "%s", p\_tinfo->*name*());

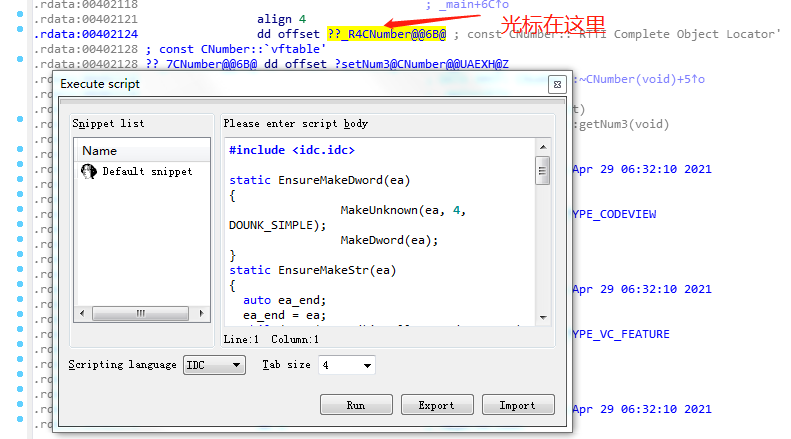
*fclose*(fp);

return 0;

return 0;

}

IDA File/Script File



这里import的GetCppRTTI.idc

#include <idc.idc>

static EnsureMakeDword(ea)

{

MakeUnknown(ea, 4, DOUNK\_SIMPLE);

MakeDword(ea);

}

static EnsureMakeStr(ea)

{

auto ea\_end;

ea\_end = ea;

while(Byte(ea\_end)!=0 && ea\_end-ea<=255) ++ea\_end;

++ea\_end; // append tial zero.

MakeUnknown(ea, ea\_end-ea, DOUNK\_SIMPLE);

MakeStr(ea, ea\_end);

}

static CommentRTTI(ea)

{

auto ea1;

ea1 = Dword(ea);

EnsureMakeDword(ea1);

EnsureMakeDword(ea1+4);

EnsureMakeDword(ea1+8);

EnsureMakeDword(ea1+12);

ea1 = Dword(ea1+12);

EnsureMakeDword(ea1);

EnsureMakeDword(ea1+4);

EnsureMakeStr(ea1+8);

return GetString(ea1+8, BADADDR, ASCSTR\_C);

}

static main()

{

auto str\_type, raw\_name, fp, name;

str\_type = GetLongPrm(INF\_STRTYPE);

SetLongPrm(INF\_STRTYPE, ASCSTR\_C);

raw\_name = CommentRTTI(ScreenEA());

Exec("C:\\111.exe " + raw\_name);

fp = fopen("c:\\111.txt", "r");

name = readstr(fp);

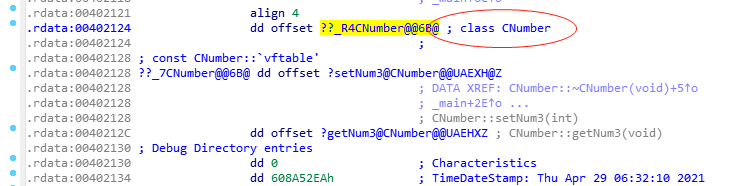
fclose(fp);

MakeRptCmt(ScreenEA(), name=="" ? raw\_name : name);

SetLongPrm(INF\_STRTYPE, str\_type);

}

IDA自动添加好了注释



这个功能感觉有点鸡肋，这还不如直接添加注释，在虚表那一行都看到类的名称CNumber了。

如果是写一个功能，直接加好所有类的注释还比较实用。

或者找找看有没有写好开源的功能。

#############################################################

下面看看NDK下面

在ndk的sdk下面搜索typeinfo

struct \_\_type\_info\_implementations {

struct \_\_string\_impl\_base {

typedef const char\* \_\_type\_name\_t;

\_LIBCPP\_INLINE\_VISIBILITY \_LIBCPP\_ALWAYS\_INLINE

\_LIBCPP\_CONSTEXPR static const char\* \_\_type\_name\_to\_string(\_\_type\_name\_t \_\_v) \_NOEXCEPT {

return \_\_v;

}

\_LIBCPP\_INLINE\_VISIBILITY \_LIBCPP\_ALWAYS\_INLINE

\_LIBCPP\_CONSTEXPR static \_\_type\_name\_t \_\_string\_to\_type\_name(const char\* \_\_v) \_NOEXCEPT {

return \_\_v;

}

};

class \_LIBCPP\_EXCEPTION\_ABI type\_info

{

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

type\_info(const type\_info&);

protected:

typedef \_\_type\_info\_implementations::\_\_impl \_\_impl;

\_\_impl::\_\_type\_name\_t \_\_type\_name;

看下来只有一个const char\*类型的数据

sizeof(type\_info) = 8

前四个字节存的是虚表指针，后四个字节存\_\_type\_name

#include <string.h>

//#include <stdlib.h>

#include <typeinfo>

using namespace std;

struct \_fake\_typeinfo {

void\* vtbl;

//void\* name;

//char raw\_name[1024];

const char\* \_\_type\_name\_t;

\_fake\_typeinfo(const char\* rname) : vtbl(nullptr) {

//strcpy(raw\_name, rname);

\_\_type\_name\_t = rname;

}

};

int main(int argc, char\*\* argv){

printf("rtti %d, %d, %d, %d\n",

\_LIBCPP\_HAS\_MERGED\_TYPEINFO\_NAMES\_DEFAULT,

sizeof(\_\_type\_info\_implementations::\_\_string\_impl\_base),

sizeof(\_\_type\_info\_implementations),

sizeof(type\_info));

const char\* decoratedName = "\_ZTS7CNumber";

\_fake\_typeinfo ti(decoratedName);

type\_info \*p\_tinfo = reinterpret\_cast<type\_info \*>(&ti);

printf("%s\n", p\_tinfo->name());

return 0;

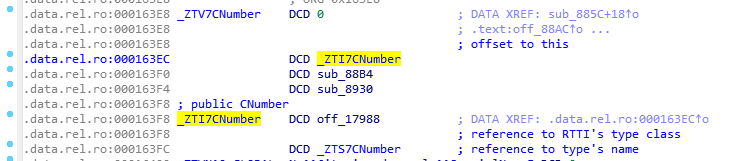
}

输出的还是\_ZTS7CNumber???

在IDA中查看：

.rodata:000064DD \_ZTS7CNumber DCB "7CNumber",0 ; DATA XREF: .data.rel.ro:000164AC↓o

IDA中看到rtti是在虚表指针后面





测试typeid

#include <string.h>

#include <typeinfo>

using namespace std;

class CNumber {

public:

CNumber() {

//printf("CNumber\n");

}

~CNumber() {

//printf("~CNumber\n");

}

void setNum1(int n) {

num1 = n;

}

int getNum1() {

return num1;

}

int getNum2() {

return num2;

}

virtual void setNum3(int n) {

num3 = n;

}

virtual int getNum3() {

return num3;

}

public:

int num1;

int num2;

int num3;

};

struct \_fake\_typeinfo {

void\* vtbl;

const char\* \_\_type\_name\_t;

\_fake\_typeinfo(const char\* rname) : vtbl(nullptr) {

\_\_type\_name\_t = rname;

}

};

class Base {};

int main(int argc, char\*\* argv){

printf("bool : %s\n", typeid(bool).name());

printf("char : %s\n", typeid(char).name());

printf("short : %s\n", typeid(short).name());

printf("int : %s\n", typeid(int).name());

printf("long : %s\n", typeid(long).name());

printf("float : %s\n", typeid(float).name());

printf("double : %s\n", typeid(double).name());

CNumber cnumber;

const type\_info& info = typeid(cnumber);

printf("%s\n", info.name());

Base b;

printf("typeid(b) : %s\n", typeid(b).name());

printf("rtti %d, %d, %d, %d, %d\n",

\_LIBCPP\_HAS\_MERGED\_TYPEINFO\_NAMES\_DEFAULT,

sizeof(\_\_type\_info\_implementations::\_\_string\_impl\_base),

sizeof(\_\_type\_info\_implementations),

sizeof(type\_info),

sizeof(\_fake\_typeinfo));

const char\* decoratedName = "7CNumber";

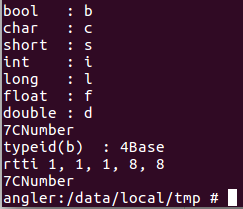
\_fake\_typeinfo ti(decoratedName);

type\_info \*p\_tinfo = reinterpret\_cast<type\_info \*>(&ti);

printf("%s\n", p\_tinfo->name());

return 0;

}



######################

查了下相关开源的插件

1、类通知器：使用C ++ RTTI和MFC RTCI类型数据扫描具有vftables的MSVC 32位目标IDB。

放置结构defs，名称，标签和注释，以更好地理解类vftables（“虚函数表”），并使它们更容易阅读，以帮助逆向工程。

创建一个列表窗口，其中包含用于浏览的已找到的vftables。

<https://sourceforge.net/projects/classinformer/>

2、Virtuailor：Virtuailor是一个IDAPython工具，可以重建用于为英特尔架构和32位和64位代码编写的C ++代码的vtable。

https://github.com/0xgalz/Virtuailor

相关视频

<https://www.youtube.com/watch?v=Xk75TM7NmtA>

这个感觉靠谱点

使用的一个视频

<https://www.youtube.com/watch?v=GEtpznMFsEs>

3、<https://github.com/nektra/vtbl-ida-pro-plugin>