C++ TUTORIAL - DYNAMIC CAST - 2020

Ph.D. / Golden Gate Ave, San Francisco / Seoul National Univ / Carnegie Mellon / UC Berkeley / DevOps / Deep Learning / Visualization

(http://www.addthis.com/bookmark.php?v=250&username=khhong7)

Sponsor Open Source development activities and free contents for everyone.

RTTI

RTTI is short for **Run-time Type Identification**. RTTI is to provide a standard way for a program to determine the type of object during runtime.

In other words, RTTI allows programs that use pointers or references to base classes to retrieve the actual derived types of the objects to which these pointers or references refer.

RTTI is provided through two operators:

1. The **typeid** operator, which returns the actual type of the object referred to by a pointer (or a reference).
2. The **dynamic_cast** operator, which safely converts from a pointer (or reference) to a base type to a pointer (or reference) to a derived type.

The dynamic_cast Operator

An attempt to **convert** an object **into** a **more specific** object.

Let's look at the code. If you do not understand what's going on, please do not worry, we'll get to it later.

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

class A
{
public:
        virtual void f(){cout << "A::f()" << endl;}
};

class B : public A
{
public:
        void f(){cout << "B::f()" << endl;}
};

int main()
{
        A a;
        B b;
        a.f();        // A::f()
        b.f();        // B::f()

        A *pA = &a;
        B *pB = &b;
        pA->f();      // A::f()
        pB->f();      // B::f()

        pA = &b;
        // pB = &a;      // not allowed
        pB = dynamic_cast<B*>(&a;); // allowed but it ret

        return 0;
}
```

The **dynamic_cast** operator is intended to be the most heavily used RTTI component. It doesn't give us what type of object a pointer points to. Instead, it answers the question of whether we can **safely assign** the address of an object to a pointer of a particular type.

cplusplus/
binarytree.php)

Blackjack with Qt (/
cplusplus/
blackjackQT.php)

Boost - shared_ptr,
weak_ptr, mpl,
lambda, etc. (/
cplusplus/
boost.php)

Boost.Asio (Socket
Programming -
Asynchronous
TCP/IP)... (/
cplusplus/Boost/
boost_AsynchIO_asio_tcpip_socket_server_client_timer_A.php)

Classes and Structs
(/cplusplus/
class.php)

Constructor (/
cplusplus/
constructor.php)

C++11(C++0x):
rvalue references,
move constructor,
and lambda, etc. (/
cplusplus/
cplusplus11.php)

C++ API Testing (/
cplusplus/

Unlike other casts, a **dynamic_cast** involves a run-time type check. If the object bound to the **pointer** is not an object of the target type, it fails and the value is **0**. If it's a **reference** type when it fails, then an exception of type **bad_cast** is thrown. So, if we want **dynamic_cast** to throw an exception (**bad_cast**) instead of returning **0**, cast to a reference instead of to a pointer. Note also that the dynamic_cast is the only cast that relies on run-time checking.

"The need for **dynamic_cast** generally arises because you want to perform derived class operation on a derived class object, but you have only a pointer or reference-to-base" said Scott Meyers in his book "Effective C++".

Let's look at the example code:

```
class Base { };

class Derived : public Base { };

int main()
{
        Base b;
        Derived d;

        Base *pb = dynamic_cast<Base*>(&d;);            /
        Derived *pd = dynamic_cast<Derived*>(&b;);      /

        return 0;
}
```

The #1 is ok because dynamic_cast is always successful when we cast a class to one of its base classes

The #2 conversion has a compilation error:

cpptesting.php)

C++ Keywords -
const, volatile, etc.
(/cplusplus/
cplusplus_keywords.php)

Debugging Crash &
Memory Leak (/
cplusplus/
CppCrashDebuggingMemoryLeak.php)

Design Patterns in
C++ ... (/
DesignPatterns/
introduction.php)

Dynamic Cast
Operator (/
cplusplus/
dynamic_cast.php)

Eclipse CDT / JNI
(Java Native
Interface) / MinGW
(/cplusplus/
eclipse_CDT_JNI_MinGW_64bit.php)

Embedded
Systems
Programming I -
Introduction (/
cplusplus/
embeddedSystemsProgramming.php)

Embedded
Systems

```
 error C2683: 'dynamic_cast' : 'Base' is not a polymorphi
```

It's because base-to-derived conversions are not allowed with dynamic_cast unless the base class is **polymorphic**.

So, if we make the Base class polymorphic by adding virtual function as in the code sample below, it will be compiled successfully.

```
class Base {virtual void vf(){}};

class Derived : public Base { };

int main()
{
        Base b;
        Derived d;

        Base *pb = dynamic_cast<Base*>(&d;);            /
        Derived *pd = dynamic_cast<Derived*>(&b;);      /

        return 0;
}
```

But at runtime, the #2 cast fails and produces **null pointer**.

Let's look at another example.

Programming II - gcc ARM Toolchain and Simple Code on Ubuntu and Fedora (/cplusplus/embeddedSystemsProgramming_gnu_toolchain_ARM_cross_co

Embedded Systems Programming III - Eclipse CDT Plugin for gcc ARM Toolchain (/cplusplus/embeddedSystemsProgramming_GNU_ARM_ToolChain_Eclipse

Exceptions (/cplusplus/exceptions.php)

Friend Functions and Friend Classes (/cplusplus/friendclass.php)

fstream: input & output (/cplusplus/fstream_input_output.php)

Function Overloading (/cplusplus/function_overloading.php)

Functors (Function Objects) I - Introduction (/

```
class Base { virtual void vf(){} };

class Derived : public Base { };

int main()
{
        Base *pBDerived = new Derived;
        Base *pBBase = new Base;
        Derived *pd;

        pd = dynamic_cast<Derived*>(pBDerived); #1
        pd = dynamic_cast<Derived*>(pBBase);    #2

        return 0;
}
```

The example has two dynamic casts from pointers of type **Base** to a point of type **Derived**. But only the #1 is successful.

Even though **pBDerived** and **pBBase** are pointers of type **Base***, **pBDerived** points to an object of type **Derived**, while **pBBase** points to an object of type **Base**. Thus, when their respective type-castings are performed using **dynamic_cast**, **pBDerived** is pointing to a full object of class **Derived**, whereas **pBBase** is pointing to an object of class **Base**, which is an incomplete object of class **Derived**.

In general, the expression

```
dynamic_cast<Type *>(ptr)
```

converts the pointer **ptr** to a pointer of type **Type*** if the pointer-to object (*ptr) is of type **Type** or else derived directly or indirectly from type **Type**. Otherwise, the expression evaluates to **0**, the null pointer.

cplusplus/functor_function_object_stl_intro.php)

Functors (Function Objects) II - Converting function to functor (/cplusplus/functor_function_object_stl_2.php)

Functors (Function Objects) - General (/cplusplus/functors.php)

Git and GitHub Express... (/cplusplus/Git/Git_GitHub_Express.php)

GTest (Google Unit Test) with Visual Studio 2012 (/cplusplus/google_unit_test_gtest.php)

Inheritance & Virtual Inheritance (multiple inheritance) (/cplusplus/multipleinheritance.php)

Libraries - Static, Shared (Dynamic)

Dynamic_cast - example

In the code below, there is one function call in main() that's not working. Which one?

(/cplusplus/libraries.php)

Linked List Basics (/cplusplus/linked_list_basics.php)

Linked List Examples (/cplusplus/linkedlist.php)

make & CMake (/cplusplus/make.php)

make (gnu) (/cplusplus/gnumake.php)

Memory Allocation (/cplusplus/memoryallocation.php)

Multi-Threaded Programming - Terminology - Semaphore, Mutex, Priority Inversion etc. (/cplusplus/multithreaded.php)

Multi-Threaded Programming II - Native Thread for Win32 (A) (/

```
#include <iostream>

using namespace std;

class A
{
public:
        virtual void g(){}
};
class B : public A
{
public:
        virtual void g(){}
};
class C : public B
{
public:
        virtual void g(){}
};
class D : public C
{
public:
        virtual void g(){}
};

A* f1()
{
        A *pa = new C;
        B *pb = dynamic_cast<B*>(pa);
        return pb;
}

A* f2()
{
        A *pb = new B;
        C *pc = dynamic_cast<C*>(pb);
        return pc;
}

A* f3()
{
        A *pa = new D;
        B *pb = dynamic_cast<B*>(pa);
```

cplusplus/multithreading_win32A.php)

Multi-Threaded Programming II - Native Thread for Win32 (B) (/cplusplus/multithreading_win32B.php)

Multi-Threaded Programming II - Native Thread for Win32 (C) (/cplusplus/multithreading_win32C.php)

Multi-Threaded Programming II - C++ Thread for Win32 (/cplusplus/multithreading_win32.php)

Multi-Threaded Programming III - C/C++ Class Thread for Pthreads (/cplusplus/multithreading_pthread.php)

MultiThreading/ Parallel Programming - IPC (/cplusplus/multithreading_ipc.php)

```
        return pb;
}

int main()
{
    f1()->g();   // (1)
    f2()->g();   // (2)
    f3()->g();   // (3)

    return 0;
}
```

Answer (2). It's a downcasting.

Dynamic_cast - another example

In this example, the **DoSomething(Window* w)** is passed down **Window** pointer. It calls **scroll()** method which is only available from **Scroll** object. So, in this case, we need to check if the object is the **Scroll** type or not before the call to the **scroll()** method.

Multi-Threaded Programming with C++11 Part A (start, join(), detach(), and ownership) (/cplusplus/multithreaded4_cplusplus11.php)

Multi-Threaded Programming with C++11 Part B (Sharing Data - mutex, and race conditions, and deadlock) (/cplusplus/multithreaded4_cplusplus11B.php)

Multithread Debugging (/cplusplus/multithreadedDebugging.php)

Object Returning (/cplusplus/object_returning.php)

Object Slicing and Virtual Table (/cplusplus/slicing.php)

OpenCV with C++ (/cplusplus/opencv.php)

Operator

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

class Window
{
public:
        Window(){}
        Window(const string s):name(s) {};
        virtual ~Window() {};
        void getName() { cout << name << endl;};
private:
        string name;
};

class ScrollWindow : public Window
{
public:
        ScrollWindow(string s) : Window(s) {};
        ~ScrollWindow() {};
        void scroll() { cout << "scroll()" << endl;};
};

void DoSomething(Window *w)
{
        w->getName();
        // w->scroll();  // class "Window" has no member

        // check if the pointer is pointing to a scroll w
        ScrollWindow *sw = dynamic_cast<ScrollWindow*>(w)

        // if not null, it's a scroll window object
        if(sw) sw->scroll();
}

int main()
{
        Window *w = new Window("plain window");
        ScrollWindow *sw = new ScrollWindow("scroll windo

        DoSomething(w);
        DoSomething(sw);
```

Overloading I (/cplusplus/operatoroverloading.php)

Operator Overloading II - self assignment (/cplusplus/operator_oveloading_self_assignment.php)

Pass by Value vs. Pass by Reference (/cplusplus/valuevsreference.php)

Pointers (/cplusplus/pointers.php)

Pointers II - void pointers & arrays (/cplusplus/pointers2_voidpointers_arrays.php)

Pointers III - pointer to function & multi-dimensional arrays (/cplusplus/pointers3_function_multidimensional_arrays.php)

Preprocessor - Macro (/cplusplus/preprocessor_macro.php)

Private Inheritance (/cplusplus/

```
        return 0;
}
```

Upcasting and Downcasting

Converting a derived-class reference or pointer to a base-class reference or pointer is called **upcasting**. It is always allowed for public inheritance without the need for an explicit type cast.

Actually this rule is part of expressing the **is-a** relationship. A **Derived** object is a **Base** object in that it inherits all the data members and member functions of a **Base** object. Thus, anything that we can do to a **Base** object, we can do to a **Derived** class object.

The **downcasting**, the opposite of upcasting, is a process converting a base-class pointer or reference to a derived-class pointer or reference.

It is not allowed without an explicit type cast. That's because a derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

Here is a self explanatory example

private_inheritance.php)

Python & C++ with SIP (/python/python_cpp_sip.php)

(Pseudo)-random numbers in C++ (/cplusplus/RandomNumbers.php)

References for Built-in Types (/cplusplus/references.php)

Socket - Server & Client (/cplusplus/sockets_server_client.php)

Socket - Server & Client 2 (/cplusplus/sockets_server_client_2.php)

Socket - Server & Client 3 (/cplusplus/sockets_server_client_3.php)

Socket - Server & Client with Qt (Asynchronous / Multithreading / ThreadPool etc.) (/cplusplus/sockets_server_client_QT.php)

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

class Employee {
private:
        int id;
public:
        void show_id(){}
};

class Programmer : public Employee {
public:
        void coding(){}
};

int main()
{
        Employee employee;
        Programmer programmer;

        // upcast - implicit upcast allowed
        Employee *pEmp = &programmer;

        // downcast - explicit type cast required
        Programmer *pProg = (Programmer *)&employee;


        // Upcasting: safe - progrommer is an Employee
        // and has his id to do show_id().
        pEmp->show_id();
        pProg->show_id();

        // Downcasting: unsafe - Employee does not have
        // the method, coding().
       // compile error: 'coding' : is not a member of 'E
        // pEmp->coding();
        pProg->coding();

        return 0;
}
```

More on Upcasting and Downcasting

Stack Unwinding (/cplusplus/stackunwinding.php)

Standard Template Library (STL) I - Vector & List (/cplusplus/stl_vector_list.php)

Standard Template Library (STL) II - Maps (/cplusplus/stl2_map.php)

Standard Template Library (STL) II - unordered_map (/cplusplus/stl2_unorderd_map_cpp11_hash_table_hash_function.php)

Standard Template Library (STL) II - Sets (/cplusplus/stl2B_set.php)

Standard Template Library (STL) III - Iterators (/cplusplus/stl3_iterators.php)

Standard Template Library (STL) IV - Algorithms (/cplusplus/

(upcasting_downcasting.php)

The typeid

typeid operator allows us to determine whether two objects are the same type.

In the previous example for Upcasting and Downcasting, **employee** gets the method **coding()** which is not desirable. So, we need to check if a pointer is pointing to the Programmer object before we use the method, coding().

Here is a new code showing how to use **typeid**:

stl4_algorithms.php)

Standard Template Library (STL) V - Function Objects (/cplusplus/stl5_function_objects.php)

Static Variables and Static Class Members (/cplusplus/statics.php)

String (/cplusplus/string.php)

String II - sstream etc. (/cplusplus/string2.php)

Taste of Assembly (/cplusplus/assembly.php)

Templates (/cplusplus/templates.php)

Template Specialization (/cplusplus/template_specialization_function_class.php)

Template Specialization - Traits (/cplusplus/

```
class Employee {
private:
        int id;
public:
        void show_id(){}
};

class Programmer : public Employee {
public:
        void coding(){}
};

#include <typeinfo>

int main()
{
        Employee lee;
        Programmer park;

        Employee *pEmpA = &lee;
        Employee *pEmpB = &park;

        // check if two object is the same
        if(typeid(Programmer) == typeid(lee)) {
                Programmer *pProg = (Programmer *)&lee;
                pProg->coding();
        }
        if(typeid(Programmer) == typeid(park)) {
                Programmer *pProg = (Programmer *)&park;
                pProg->coding();
        }

        pEmpA->show_id();
        pEmpB->show_id();

        return 0;
}
```

So, only a programmer uses the **coding()** method.

Note that we included <typeinfo> in the example. The **typeid**

template_specialization_traits.php)

Template Implementation & Compiler (.h or .cpp?) (/cplusplus/template_declaration_definition_header_implementation_file.p

The this Pointer (/cplusplus/this_pointer.php)

Type Cast Operators (/cplusplus/typecast.php)

Upcasting and Downcasting (/cplusplus/upcasting_downcasting.php)

Virtual Destructor & boost::shared_ptr (/cplusplus/virtual_destructors_shared_ptr.php)

Virtual Functions (/cplusplus/virtualfunctions.php)

Programming Questions and Solutions ↓

operator returns a reference to a **type_info** object, where **type_info** is a class defined in the **typeinfo** header file.

RTTI - pros and cons

This is from Google C++ Style Guide (http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Run-Time_Type_Information__RTTI_).

RTTI allows a programmer to query the C++ class of an object at run time. This is done by use of **typeid** or **dynamic_cast**. Avoid using Run Time Type Information (RTTI).

1. **Pros**
   The standard alternatives to RTTI (described below) require modification or redesign of the class hierarchy in question. Sometimes such modifications are infeasible or undesirable, particularly in widely-used or mature code.
   RTTI can be useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type. It is also useful in managing the relationship between objects and their mocks.
   RTTI is useful when considering multiple abstract objects. Consider

Strings and Arrays (/cplusplus/quiz_strings_arrays.php)

Linked List (/cplusplus/quiz_linkedlist.php)

Recursion (/cplusplus/quiz_recursion.php)

Bit Manipulation (/cplusplus/quiz_bit_manipulation.php)

Small Programs (string, memory functions etc.) (/cplusplus/smallprograms.php)

Math & Probability (/cplusplus/quiz_math_probability.php)

Multithreading (/cplusplus/quiz_multithreading.php)

140 Questions by Google (/cplusplus/google_interview_questions.php)

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast&LT;Derived*&GT;(other)
  if (that == NULL)
    return false;
  ...
}
```

2. **Cons**
   Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed.
   Undisciplined use of RTTI makes code hard to maintain. It can lead to type-based decision trees or switch statements scattered throughout the code, all of which must be examined when making further changes.

3. **Decision**
   RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unittests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code. If you find yourself needing to write code that behaves differently based on the class of an object, consider one of the following alternatives to querying the type:
   1. Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
   2. If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

When the logic of a program guarantees that a given instance of a base class is in fact an instance of a particular derived class, then a **dynamic_cast** may be used freely on the object. Usually one can use a **static_cast** as an alternative in such situations.
Decision trees based on type are a strong indication that your code is on the wrong track.

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

Code such as this usually breaks when additional subclasses are added to the class hierarchy. Moreover, when properties of a subclass change, it is difficult to find and modify all the affected code segments.
Do not hand-implement an RTTI-like workaround. The arguments against RTTI apply just as much to workarounds like class hierarchies with type tags. Moreover, workarounds disguise your true intent.

CONTACT

BogoToBogo
contactus@bogotobogo.com (mailto:contactus@bogotobogo.com)

FOLLOW BOGOTOBOGO

(https://www.facebook.com/KHongSanFrancisco)
(https://twitter.com/KHongTwit)

ABOUT US (/ABOUT_US.PHP)

contactus@bogotobogo.com (mailto:contactus@bogotobogo.com)

Pacific Ave, San Francisco, CA 94115

Pacific Ave, San Francisco, CA 94115