25.10 — Dynamic casting

ALEX[1] DECEMBER 29, 2024

Way back in lesson 10.6 -- Explicit type conversion (casting) and static_cast (https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/)[2], we examined the concept of casting, and the use of static_cast to convert variables from one type to another.

In this lesson, we'll continue by examining another type of cast: dynamic_cast.

The need for dynamic_cast

When dealing with polymorphism, you'll often encounter cases where you have a pointer to a base class, but you want to access some information that exists only in a derived class.

Consider the following (slightly contrived) program:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Base
{
protected:
    int m_value{};

public:
    Base(int value)
        : m_value{value}
    {
    }

    virtual ~Base() = default;
};

class Derived : public Base
{
protected:
    std::string m_name{};

public:
    Derived(int value, std::string_view name)
        : Base{value}, m_name{name}
    {
    }

    const std::string& getName() const { return m_name; }
};

Base* getObject(bool returnDerived)
{
    if (returnDerived)
        return new Derived{1, "Apple"};
    else
        return new Base{2};
}

int main()
{
    Base* b{ getObject(true) };

    // how do we print the Derived object's name here, having only a Base pointer?

    delete b;

    return 0;
}
```

In this program, function getObject() always returns a Base pointer, but that pointer may be pointing to either a Base or a Derived object. In the case where the Base pointer is actually pointing to a Derived object, how would we call Derived::getName()?

One way would be to add a virtual function to Base called getName() (so we could call it with a Base pointer/reference, and have it dynamically resolve to Derived::getName()). But what would this function return if you called it with a Base pointer/reference that was actually pointing to a Base object? There isn't really any value that makes sense. Furthermore, we would be polluting our Base class with things that really should only be the concern of the Derived class.

We know that C++ will implicitly let you convert a Derived pointer into a Base pointer (in fact, getObject() does just that). This process is sometimes called **upcasting**. However, what if there was a way to convert a

Base pointer back into a Derived pointer? Then we could call Derived::getName() directly using that pointer, and not have to worry about virtual function resolution at all.

dynamic_cast

C++ provides a casting operator named **dynamic_cast** that can be used for just this purpose. Although dynamic casts have a few different capabilities, by far the most common use for dynamic casting is for converting base-class pointers into derived-class pointers. This process is called **downcasting**.

Using dynamic_cast works just like static_cast. Here's our example main() from above, using a dynamic_cast to convert our Base pointer back into a Derived pointer:

```
int main()
{
    Base* b{ getObject(true) };

    Derived* d{ dynamic_cast<Derived*>(b) }; // use dynamic cast to convert Base
pointer into Derived pointer

    std::cout << "The name of the Derived is: " << d->getName() << '\n';

    delete b;

    return 0;
}
```

This prints:

```
The name of the Derived is: Apple
```

dynamic_cast failure

The above example works because b is actually pointing to a Derived object, so converting b into a Derived pointer is successful.

However, we've made quite a dangerous assumption: that b is pointing to a Derived object. What if b wasn't pointing to a Derived object? This is easily tested by changing the argument to getObject() from true to false. In that case, getObject() will return a Base pointer to a Base object. When we try to dynamic_cast that to a Derived, it will fail, because the conversion can't be made.

If a dynamic_cast fails, the result of the conversion will be a null pointer.

Because we haven't checked for a null pointer result, we access d->getName(), which will try to dereference a null pointer, leading to undefined behavior (probably a crash).

In order to make this program safe, we need to ensure the result of the dynamic_cast actually succeeded:

```cpp
int main()
{
    Base* b{ getObject(true) };

    Derived* d{ dynamic_cast<Derived*>(b) }; // use dynamic cast to convert Base
pointer into Derived pointer

    if (d) // make sure d is non-null
        std::cout << "The name of the Derived is: " << d->getName() << '\n';

    delete b;

    return 0;
}
```

Rule

Always ensure your dynamic casts actually succeeded by checking for a null pointer result.

Note that because dynamic_cast does some consistency checking at runtime (to ensure the conversion can be made), use of dynamic_cast does incur a performance penalty.

Also note that there are several cases where downcasting using dynamic_cast will not work:

1. With protected or private inheritance.
2. For classes that do not declare or inherit any virtual functions (and thus don't have a virtual table).
3. In certain cases involving virtual base classes (see this page[3] for an example of some of these cases, and how to resolve them).

Downcasting with static_cast

It turns out that downcasting can also be done with static_cast. The main difference is that static_cast does no runtime type checking to ensure that what you're doing makes sense. This makes using static_cast faster, but more dangerous. If you cast a Base* to a Derived*, it will "succeed" even if the Base pointer isn't pointing to a Derived object. This will result in undefined behavior when you try to access the resulting Derived pointer (that is actually pointing to a Base object).

If you're absolutely sure that the pointer you're downcasting will succeed, then using static_cast is acceptable. One way to ensure that you know what type of object you're pointing to is to use a virtual function. Here's one (not great) way to do that:

```cpp
#include <iostream>
#include <string>
#include <string_view>

// Class identifier
enum class ClassID
{
    base,
    derived
    // Others can be added here later
};

class Base
{
protected:
    int m_value{};

public:
    Base(int value)
        : m_value{value}
    {
    }

    virtual ~Base() = default;
    virtual ClassID getClassID() const { return ClassID::base; }
};

class Derived : public Base
{
protected:
    std::string m_name{};

public:
    Derived(int value, std::string_view name)
        : Base{value}, m_name{name}
    {
    }

    const std::string& getName() const { return m_name; }
    ClassID getClassID() const override { return ClassID::derived; }

};

Base* getObject(bool bReturnDerived)
{
    if (bReturnDerived)
        return new Derived{1, "Apple"};
    else
        return new Base{2};
}

int main()
{
    Base* b{ getObject(true) };

    if (b->getClassID() == ClassID::derived)
    {
        // We already proved b is pointing to a Derived object, so this should always
succeed
        Derived* d{ static_cast<Derived*>(b) };
        std::cout << "The name of the Derived is: " << d->getName() << '\n';
    }

    delete b;

    return 0;
}
```

But if you're going to go through all of the trouble to implement this (and pay the cost of calling a virtual function and processing the result), you might as well just use dynamic_cast.

Also consider what would happen if our object were actually some class that is derived from Derived (let's call it `D2`). The above check `b->getClassID() == ClassID::derived` will fail because `getClassId()` would return `ClassID::D2`, which is not equal to `ClassID::derived`. Dynamic casting `D2` to `Derived` would succeed though, since a `D2` is a `Derived`!

dynamic_cast and references

Although all of the above examples show dynamic casting of pointers (which is more common), dynamic_cast can also be used with references. This works analogously to how dynamic_cast works with pointers.

```
#include <iostream>
#include <string>
#include <string_view>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{value}
    {
    }

    virtual ~Base() = default;
};

class Derived : public Base
{
protected:
    std::string m_name;

public:
    Derived(int value, std::string_view name)
        : Base{value}, m_name{name}
    {
    }

    const std::string& getName() const { return m_name; }
};

int main()
{
    Derived apple{1, "Apple"}; // create an apple
    Base& b{ apple }; // set base reference to object
    Derived& d{ dynamic_cast<Derived&>(b) }; // dynamic cast using a reference instead
of a pointer

    std::cout << "The name of the Derived is: " << d.getName() << '\n'; // we can
access Derived::getName through d

    return 0;
}
```

Because C++ does not have a "null reference", dynamic_cast can't return a null reference upon failure. Instead, if the dynamic_cast of a reference fails, an exception of type std::bad_cast is thrown. We talk about exceptions later in this tutorial.

dynamic_cast vs static_cast

New programmers are sometimes confused about when to use static_cast vs dynamic_cast. The answer is quite simple: use static_cast unless you're downcasting, in which case dynamic_cast is usually a better choice. However, you should also consider avoiding casting altogether and just use virtual functions.

Downcasting vs virtual functions

There are some developers who believe dynamic_cast is evil and indicative of a bad class design. Instead, these programmers say you should use virtual functions.

In general, using a virtual function *should* be preferred over downcasting. However, there are times when downcasting is the better choice:

- When you can not modify the base class to add a virtual function (e.g. because the base class is part of the standard library)
- When you need access to something that is derived-class specific (e.g. an access function that only exists in the derived class)
- When adding a virtual function to your base class doesn't make sense (e.g. there is no appropriate value for the base class to return). Using a pure virtual function may be an option here if you don't need to instantiate the base class.

A warning about dynamic_cast and RTTI

Run-time type information (RTTI) is a feature of C++ that exposes information about an object's data type at runtime. This capability is leveraged by dynamic_cast. Because RTTI has a pretty significant space performance cost, some compilers allow you to turn RTTI off as an optimization. Needless to say, if you do this, dynamic_cast won't function correctly.

Next lesson

25.11 Printing inherited classes using operator<<

4

Back to table of contents

5

Previous lesson

25.9 Object slicing

6

7

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[8] are connected to your provided email address.

141 COMMENTS

Newest

Quizzzzeeeezsss

June 27, 2025 7:07 am PDT

More quizzes, please.

0 Reply

KALEiDOSCOPE

April 21, 2025 7:47 am PDT

Say we have a vector containing Base, *and I've put in many addresses of Derived1, Derived2, Derived3, Derived4, ... How do I know which one Base* is pointing to?

0 Reply

KALEiDOSCOPE

Reply to KALEiDOSCOPE [9] April 21, 2025 7:48 am PDT

It's Base*. Asteriks made it italicized.

0 Reply

Amin

February 26, 2024 2:28 am PST

Lets say we have this situation:

```cpp
class Base
{
protected:
    int m_value{};

public:
    Base(int value)
        : m_value{value}
    {
    }

    virtual ~Base() = default;
};

class Derived1 : public Base
{
protected:
    std::string m_name{};

public:
    Derived1(int value, std::string_view name)
        : Base{value}, m_name{name}
    {
    }

    const std::string& getName() const { return m_name; }
};

class Derived2 : public Base
{
protected:
    std::string m_name{};
    std::string m_color{};

public:
    Derived2(int value, std::string_view name, std::string_view color)
        : Base{value}, m_name{name}, m_color{color}
    {
    }

    const std::string& getColor() const { return m_color; }
};

int main()
{
    Derived2 red_apple{1, "Apple","Red"}; // create a red apple
    Derived1& d{ ???_cast<Derived1&>(&red_apple) }; // convert to Derived1 so we can call getName because Derived2 does not implement that for some important reason!

    std::cout << "The name of the Derived is: " << d.getName() << '\n';

    return 0;
}
```

what is the best way of casting this? Currently I am doing this:

```cpp
static_cast<Derived1&>(static_cast<Base&>(&red_apple))
```

0 Reply

Alex Author

Reply to Amin [10] February 27, 2024 8:10 pm PST

I think like this:

```
1 | Derived1& d{ reinterpret_cast<Derived1&>(red_apple) };
```

We want the compiler to interpret `red_apple` as if it were a Derived1.

0 Reply

Timon

Reply to Alex [11] March 27, 2024 1:14 pm PDT

Weird case it kind of works, but if you switch m_name and m_color in Derived 2 getName() will print the color.
Because m_name is not defined in Base.

```cpp
#include <iostream>

class Base
{
protected:
    int m_value{};

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual ~Base() = default;
};

class Derived1 : public Base
{
protected:
    std::string m_name{};

public:
    Derived1(int value, std::string_view name)
        : Base{ value }, m_name{ name }
    {
    }

    const std::string& getName() const { return m_name; }
};

class Derived2 : public Base
{
protected:
    std::string m_color{};
    std::string m_name{};

public:
    Derived2(int value, std::string_view name, std::string_view color)
        : Base{ value }, m_name{ name }, m_color{ color }
    {
    }

    const std::string& getColor() const { return m_color; }
};

int main()
{
    Derived2 red_apple{ 1, "Apple","Red" }; // create a red apple
    Derived1& d{ reinterpret_cast<Derived1&>(red_apple) }; // convert to Derived1 so we can call getName because Derived2 does not implement that for some important reason!

    std::cout << "The name of the Derived is: " << d.getName() << '\n';

    return 0;
}
```

The name of the Derived is: Red

is this considered undefined behavior?

0 Reply

Alex Author

Reply to Timon [12] March 29, 2024 11:25 am PDT

I don't think so, since `Derived2::m_color` and `Derived1::m_name` have the same type. But it's certainly confusing.

0 Reply

D D

December 10, 2023 9:19 am PST

Hello, I would say that using static conversion is better used for step-down conversion, even if we have multiple inheritance. because this transformation is smart enough to follow the pointer from one class to another. but if we have virtual inheritance, then it is more correct to use dynamic transformation, because it will go back and forth between virtual tables.

0 Reply

Zoltan

October 28, 2023 10:39 am PDT

Isn't this a typo?

"In the case where the pointer is pointing to a **Derived** object, how would we call Derived::getName()?"

I think this sentence was supposed to be:
"In the case where the pointer is pointing to a **Base** object, how would we call Derived::getName()?"

0 Reply

Alex Author

Reply to Zoltan [13] October 30, 2023 11:50 am PDT

No, it's correct as written. The pointer has type `Base*`, but we're interested in the case where we're actually pointing to a `Derived` object. I added a few words to the article to try to make this a bit clearer.

1 Reply

Helix

September 10, 2023 5:40 am PDT

What actually happens when you `dynamic_cast` a base pointer to a derived one?

The base pointer is allowed to walk through only the base portion of the derived object; but, upon dynamic casting, a new pointer is returned which has the capability to walk through the entire derived object?

Last edited 1 year ago by Helix

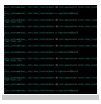
1 Reply

Alex Author

Reply to Helix [14] September 13, 2023 1:38 pm PDT

Yes, a derived pointer has full access to the derived object.

5 Reply



learnccp lesson reviewer

July 26, 2023 7:59 am PDT

W LESSON, WHERE QUIZZES

1 Reply

evilbabaroga

July 12, 2023 10:01 am PDT

yo where the quizzes go? i cant tell if i remember anything i learn without them!

1 Reply

noctis

July 9, 2023 8:09 am PDT

Can you plese help here. These are the following reasons as mentioned above for using dynamic_cast.

```
there are times when downcasting is the better choice:

    1. When you can not modify the base class to add a virtual function (e.g. because the base class is part of the standard library)
    2. When you need access to something that is derived-class specific (e.g. an access function that only exists in the derived class)
    3. When adding a virtual function to your base class doesn't make sense (e.g. there is no appropriate value for the base class to return). Using a pure virtual function may be an option here if you don't need to instantiate the base class.
```

1st and 3rd even 2nd (slightly) talks about not using or able to use `virtual` functionality. But then we have this -

```
Also note that there are several cases where downcasting using dynamic_cast will not work:

  1.  With protected or private inheritance.
  2.  For classes that do not declare or inherit any virtual functions (and thus don't have a virtual table).
  3.  In certain cases involving virtual base classes (see this page for an example of some of these cases, and how to resolve them).
```

In 2nd point `virtual` is mandotory. why this limit ?

0 Reply

Alex Author

Reply to noctis [15] July 9, 2023 8:09 pm PDT

Because C++ has a philosophy that you shouldn't pay for what you don't use.

Virtual function resolution requires that an object have a virtual table, which takes up memory. Therefore, if your object doesn't have any virtual functions defined, there won't be a virtual table, and dynamic_cast won't work.

It is possible to have a class that uses inheritance but not any virtual functions.

0 Reply

sydney

July 1, 2023 10:47 am PDT

I using DevC++ and when I go to compile your code, I get "[Error] string_view: No such file or directory"! Most of your code examples do work, but not these on dynamic_cast since they all use the string_view library.

0 Reply

Alex Author

Reply to sydney [16] July 3, 2023 5:37 pm PDT

Sounds like your version of DevC++ doesn't support C++17.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/
3. https://msdn.microsoft.com/en-us/library/cby9kycs.aspx
4. https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/object-slicing/
7. https://www.learncpp.com/dynamic-casting/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-609436
10. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-594049
11. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-594117
12. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-595172
13. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-589188

14. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-586912
15. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-583463
16. https://www.learncpp.com/cpp-tutorial/dynamic-casting/#comment-582943
17. https://g.ezoic.net/privacy/learncpp.com