

25.10 — Dynamic casting

 **ALEX**  **OCTOBER 30, 2023**

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/>), 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:

```
#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:

```
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](#) 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:

```

#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; }
    virtual ClassID getClassID() const { 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<<](#)



Back to table of contents



Previous lesson

25.9 [Object slicing](#)

Leave a comment...



Name*



Email*



Find a mistake? Leave a comment above!?



Avatars from <https://gravatar.com/> are connected to your provided email address.

Notify me about replies:



POST COMMENT

134 COMMENTS

Newest ▾

We and our partners share information on your use of this website to help improve your experience.

Do not sell my info:



OKAY

×

