

Tutorial – Implicit Conversions and Casting

Introduction:

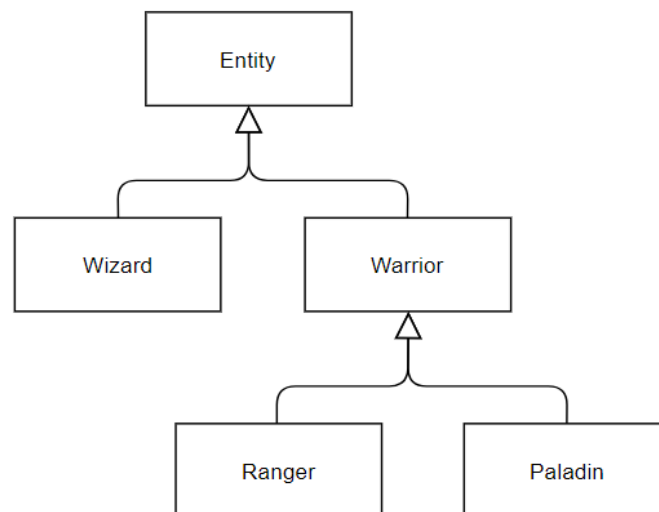
In this tutorial, we are going to look at dynamic casting by making a very simple program that highlights its use.

We will then discuss some of the pros and cons of the *dynamic_cast* operator.

Wizards Attack!:

Let's imagine that we've been handed a freshly minted design document to implement. The document describes some sort of Dungeons and Dragons game, and details a character class hierarchy that includes wizards, rangers, paladins, along with a host of other classes.

Being well versed in the object-oriented methodology, we've naturally turned this into the following class hierarchy:



This simple class diagram shows that we have several classes defined, all of them being derived from a common base class called *Entity*. *Ranger* and *Paladin* are sub-classes of *Warrior*, giving us an additional layer of inheritance.

In our game, all of these entities can attack, so we'll define a virtual function in the *Entity* base called *attack()*.

Here's what our design looks like in code:

```
#include <iostream>

class Entity {
public:
    Entity() {};
    virtual ~Entity() {};

    virtual void attack() = 0;
};

class Warrior : public Entity {
public:
    Warrior() {};
    virtual ~Warrior() {};

    virtual void attack() = 0;
};

class Ranger : public Warrior {
public:
    Ranger() {};
    ~Ranger() {};

    void attack() { std::cout << "Back, foul beast!" << std::endl; }
};

class Paladin : public Warrior {
public:
    Paladin() {};
    ~Paladin() {};

    void attack() { std::cout << "For the glory!" << std::endl; }
};

class Wizard : public Entity {
public:
    Wizard() {};
    ~Wizard() {};

    void attack() { std::cout << "You shall not pass!" << std::endl; }
    void heal() { std::cout << "You are revived" << std::endl; }
};
```

You can see from the code above that we've got a couple layers of inheritance and are using abstract functions to define an interface that sub-classes must implement.

In a real game, this could be quite a complex hierarchy. Let's pretend that we're working with code that is much more complicated than it really is.

In our game (the *main()* function) we want to define a list of *Entity* pointers, where each *Entity* pointer could point to an instance of any one of these sub-classes.

Later, in our game loop, we want to step through our list of entities and all the *attack()* function on each one. The following *main()* function does exactly this (to illustrate this concept, rather than trying to implement a complete game):

```
#include <iostream>
#include <vector>

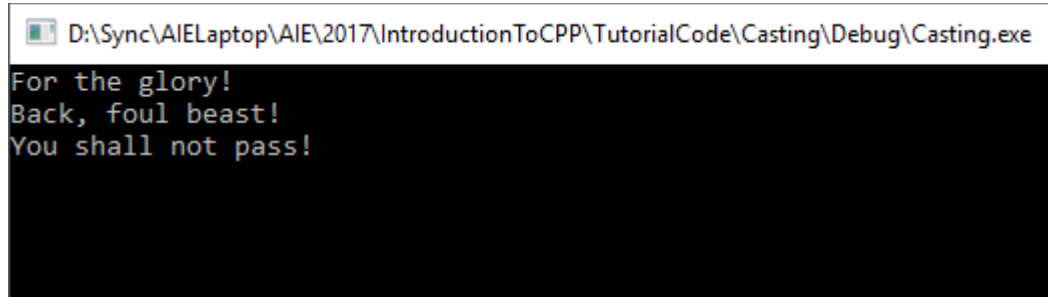
void main()
{
    std::vector<Entity*> fighters;

    fighters.push_back(new Paladin());
    fighters.push_back(new Ranger());
    fighters.push_back(new Wizard());

    for (Entity* fighter : fighters)
    {
        fighter->attack();
    }

    std::cin.get();
}
```

After executing this program we see the following output:



What did we just do? We successfully used inheritance and polymorphism to define different behaviour based on the class of each instance.

Wizards Heal?:

As development continues we get a change request. Our producer wants wizards to be able to heal.

We add the *heal()* function to the *Wizard* class, but in our *main()* function there is no way to call *heal()*.

Adding a call to *heal()* using our *Entity* pointer simply won't work:

```
for (Entity* fighter : fighters)
{
    fighter->attack();
    fighter->heal();           // you can't do this
}
```

The function *heal()* hasn't been defined in the base class, only in the *Wizard* sub-class.

The obvious solution might be to add another abstract function to the base class, but let's assume that for a variety of reasons this isn't possible (perhaps *Entity* is defined in a read-only library, or the inheritance hierarchy is already too complex and we don't wish to add another function to the base class just to support this one sub-class).

We can use the *dynamic_cast* operator to see if the *Entity* pointer *fighter* points to an instance of the *Wizard* class. If it does, then we can call the *heal()* function.

We can update our *main()* function as follows:

```
void main()
{
    std::vector<Entity*> fighters;

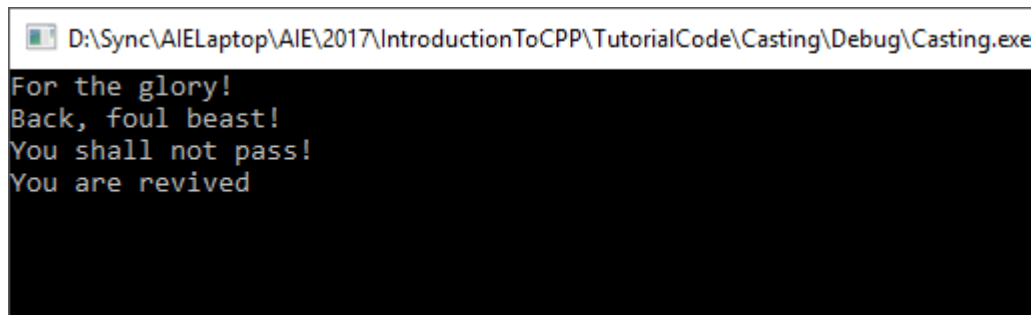
    fighters.push_back(new Paladin());
    fighters.push_back(new Ranger());
    fighters.push_back(new Wizard());

    for (Entity* fighter : fighters)
    {
        fighter->attack();

        Wizard* wiz = dynamic_cast<Wizard*>(fighter);
        if (wiz != nullptr) {
            wiz->heal();
        }
    }

    std::cin.get();
}
```

We attempt the dynamic cast on the *fighter* variable, storing the result in *wiz*. If the cast is successful (*fighter* points to an instance of the *Wizard* class), then *wiz* will be not null. We then know we can safely call the *heal()* function.



Upcasting and Downcasting:

Converting a derived class pointer or reference (i.e., *Wizard*) to a base-class pointer or reference (i.e., *Entity*) is called *upcasting*. It is always allowed for public inheritance (where the *public* keyword comes before the base-class name when declaring the sub-class) without the need for any sort of explicit type cast.

We say an example of this when we created a new instance of the *Wizard* class (and the other derived classes) and added them to the vector containing *Entity* pointers.

Downcasting is the opposite of upcasting. It is the process of converting a base-class pointer or reference to a derived-class pointer or reference.

Downcasting 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 use these data members wouldn't apply to the base class.

In the code above, downcasting was performed using the *dynamic_cast* operator.

The Pros and Cons of Dynamic Casting:

Dynamic casting uses the Run Time Type Information (RTTI) of a class to determine if it can be cast to a specific type, at run-time.

There is a bit of controversy surrounding dynamic casting. In many cases what can be achieved by the *dynamic_cast* can also be achieved through a modification or redesign of the class hierarchy. *dynamic_cast* is also prone to abuse.

The following comes from Google's C++ Style Guide:

https://google.github.io/styleguide/cppguide.html#Run-Time_Type_Information_RTTI

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.

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:

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

Decision:

RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unit tests, but avoid it when possible in other code. 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:

- 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.
- 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 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.

We cover both unit testing and the factory pattern in the subject *Code Design and Data Structures*.

The take-away point from this extract is that if you need a `dynamic_cast` to solve some problem, it probably indicates a deeper issue with the design of your program.

Other Notes:

The *dynamic_cast* operator will only work in cases where the class hierarchy has at least one virtual function. This is because the compiler uses the *vtable* (virtual method table) to determine if a cast to the desired type is possible. The *vtable* is only produced when a class defines or inherits a virtual function, and it allows the runtime to invoke the appropriate function implementations within the class hierarchy.

dynamic_cast also requires that *RTTI* (Run Time Type Information) be enabled. RTTI is optional with some compilers. It might also be turned off for a lot of custom engines.