C++ Type Erasure: Wrapping Any Type

Understanding how to write a wrapper of any type in a class in C++ to increase the level of abstraction in our code.

Debby Nirwan Oct 28, 2021 · 6 min read ★

Photo by AltumCode on Unsplash

Introduction

We start with some basics about Generic Programming, Object Oriented Programming, and Duck Typing concepts to understand what problems this technique tries to solve. We'll cover the details step-by-step at the end of this post.

Generic Programming

Generic Programming refers to a style of programming in which functions or algorithms are written to accept different types as opposed to a single type. In C++ you can achieve this with **Templates**.

Using templates we can ask the compiler to generate functions/classes for us based on the types used in our code. This helps us to not write similar blocks of code multiple times to keep our code neat. Here's a simple example:

```cpp
int Add(int a, int b)
{
  return a+b;
}
double Add(double a, double b)
{
  return a+b;
}
```

type_erasure1.cpp hosted with ♡ by GitHub view raw

Instead of writing the two functions above, we can create a template as follows.

```
1  template <typename T>
2  T Add(T a, T b)
3  {
4    return a+b;
5  }
```

type_erasure2.cpp hosted with ♡ by GitHub view raw

The compiler will generate the functions for us, only when we need them. For example when we add these calls to our **main()** function:

```
1  int main()
2  {
3    int a = Add(1, 1);
4    double b = Add(1.0, 1.0);
5    return 0;
6  }
```

type_erasure3.cpp hosted with ♡ by GitHub view raw

The compiler will generate these functions:

```
int Add<int>(int a, int b)
{
  return a+b;
}
double Add<double>(double a, double b)
{
  return a+b;
}
```

type_erasure4.cpp hosted with ♡ by GitHub view raw

Uniform Interface

Some of the basic programming principles include “Don’t Repeat Yourself — DRY”, which is to avoid duplication in our code. One of the ways to do it is by creating a uniform interface.

To understand the idea better, let’s look at an example. Say we want to create a function that receives any type that has an *Id* which we can get by calling **GetId()** function which we want to print, let’s call our function **PrintId()**.

```
void PrintId(Object obj)
{
  std::cout << obj.GetId(); << "\n";
}
```

type_erasure5.cpp hosted with ♡ by GitHub view raw

type_erasure5.cpp hosted with ♡ by GitHub view raw

Assume that we have two different types and one free function as follows, and we can add lambdas as well.

```
1  struct Object1
2  {
3    static constexpr int id = 1;
4    int GetId() const {return id;}
5  };
6
7  struct Object2
8  {
9    static constexpr int id = 2;
10   int GetId() const {return id;}
11 };
12
13 int GetId()
14 {
15   return 0;
16 }
```

type_erasure6.cpp hosted with ♡ by GitHub view raw

The naive solution is to write function overloads for each of the types that we want to print:

```
1  void PrintId(Object1 obj1)
```

```
2  {
3    std::cout << obj1.GetId() << "\n";
4  }
5
6  void PrintId(Object2 obj2)
7  {
8    std::cout << obj2.GetId() << "\n";
9  }
10
11 void PrintId(int(*obj)())
12 {
13   std::cout << obj() << "\n";
14 }
```

type_erasure7.cpp hosted with ♡ by GitHub view raw

Here's how we can invoke them:

```
1  int main()
2  {
3    Object1 obj1;
4    Object2 obj2;
5    PrintId(obj1);
6    PrintId(obj2);
7    PrintId(GetId);
8    PrintId([](){ return 4; });
9  }
```

type_erasure7.cpp hosted with ♡ by GitHub view raw

The third function can handle both free functions and **capture-less** lambdas. It works for **capture-less** lambdas because they **decay** into function pointers.

The problem that we want to solve here is that we want to have one **PrintId()** function that is able to accept all the types above.

Polymorphism

The first thing that comes to mind is to use inheritance, a concept in OOP. We can create an interface, Abstract Base Class — ABC, and let the objects inherit from or implement it.

```
1  struct Object
2  {
3    virtual int GetId() const = 0;
4  };
5
6  struct Object1 : public Object
7  {
8    static constexpr int id = 1;
9    int GetId() const override {return id;}
10 };
11
12 struct Object2 : public Object
13 {
14   static constexpr int id = 2;
15   int GetId() const override {return id;}
```

```
16  };
```

type_erasure8.cpp hosted with ♡ by GitHub view raw

And have only one **PrintId()** function implemented:

```
1   void PrintId(const Object *obj)
2   {
3     std::cout << obj->GetId() << "\n";
4   }
5
6   int main()
7   {
8     Object1 obj1;
9     Object2 obj2;
10    PrintId(&obj1);
11    PrintId(&obj2);
12    // PrintId(GetId); // doesn't work due to type incompatibility
13    // PrintId([](){ return 4; }); // same as above
14  }
```

type_erasure9.cpp hosted with ♡ by GitHub view raw

Obviously, it doesn't work when we pass different types, such as function pointers.

Templates

When talking about uniform interfaces, there is another concept from Python programming language called **Duck Typing**. It basically means that we don't need to know about the type of object as long as it supports the behavior that we need. In our case, we need a behavior/function that accepts nothing and returns an int, **int(*behavior)()**.

In C++, we can use templates to achieve the same thing. The difference with Python is that in Python the checking is at runtime whereas in C++ is at compile time.

```
1  template<typename T>
2  void PrintId(T obj)
3  {
4    std::cout << obj.GetId() << "\n";
5  }
```

type_erasure10.cpp hosted with ♡ by GitHub view raw

Now, **Object1** and **Object2** don't need to be related to each other, as long as they have a function named **GetId**, it'll work just fine. Under the hood, we actually have two different functions (see the first section about Generic Programming).

But, we still can't call free functions and lambdas, because they don't have a function named **GetId()**.

```
int main()
{
  Object1 obj1;
  Object2 obj2;
  PrintId(obj1);
  PrintId(obj2);
  // PrintId(GetId); // error: no function named GetId()
  // PrintId([](){ return 4; }); // same as above
}
```

type_erasure11.cpp hosted with ♡ by GitHub view raw

Use Functors to make it more Generic

We can change our classes and **PrintId()** function to make it more generic by using functors. Read my post about C++ Lambda if you need more details.

C++ Basics: Understanding Lambda

A convenient way to define a functor that can help us to simplify the code.

towardsdatascience.com

```cpp
struct Object1
{
  static constexpr int id = 1;
  int operator()() const {return id;}
};

struct Object2
{
  static constexpr int id = 2;
  int operator()() const {return id;}
};

template<typename T>
void PrintId(T obj)
{
  std::cout << obj() << "\n";
}
```

type_erasure12.cpp hosted with ♡ by GitHub view raw

With this change, all types work.

```cpp
int main()
{
  Object1 obj1;
  Object2 obj2;
  PrintId(obj1);
  PrintId(obj2);
  PrintId(GetId);
  PrintId(+[](){ return 4; });
  PrintId([=](){ return 4; }); // even capturing lambdas work
```

```
10  }
```

type_erasure13.cpp hosted with ♡ by GitHub view raw

Under the hood, we have many functions generated by the compiler, they are:

```
1  void PrintId<Object1>(Object1); // Object1
2  void PrintId<Object2>(Object2); // Object2
3  void PrintId<int (*)()>(int (*)()); // GetId and 1stlambda
4  void PrintId<main::{lambda()#2}>(main::{lambda()#2}):  // 2nd lambda
```

type_erasure14.cpp hosted with ♡ by GitHub view raw

The 1st lambda is implicitly converted into a function pointer by adding '+' operator in front of it.

Now it looks like we have our solution, but we still have a problem to solve.

> What if we want to store the objects into an array?

Wrapping Any Type with the same behavior

In some scenarios in our code, we'd want to store the callable objects into an array or some other containers. With our implementation so far it is not

possible because we don't have a single type required by STL containers such as **std::vector**. To give a concrete example, if we changed our **PrintId()** function to:

```cpp
void PrintId(const std::vector<ObjectWrapper>& objects)
{
  for (const auto& object : objects)
  {
    std::cout << object() << "\n";
  }
}
```

type_erasure15.cpp hosted with ♡ by GitHub view raw

we will have compile error because we have not implemented **ObjectWrapper** class.

Creating a Wrapper of any type

To solve this issue we need a single class that can wrap all our objects of different types. This is where we use the **Type Erasure** technique in C++. Let's build our wrapper step-by-step. Our wrapper class has to:

Expose common interfaces

The first thing that our wrapper needs to provide are uniform interfaces, in our case, it is ***int(*behavior)()***.

```
1  struct ObjectWrapper
2  {
3    int operator()() const;
4  };
```

type_erasure16.cpp hosted with ♡ by GitHub view raw

Make a copy of the passed objects

Our wrapper class has to manage the lifetime of the objects to avoid using dangling references or pointers. Another consideration is the size of the objects may be quite large, so we need to store them in the free store/heap. We choose to use a smart pointer. We need a new type for this, we call it **ObjectBase**.

```
1  struct ObjectWrapper
2  {
3    int operator()() const;
4    std::shared_ptr<ObjectBase> wrappedObject;
5  };
```

type_erasure17.cpp hosted with ♡ by GitHub view raw

Polymorphism

Now we have a new type called **ObjectBase** and our goal is to wrap any type. The solution is to use polymorphism, we make **ObjectBase** an interface and get help from the compiler to create child classes that inherit from it.

```
struct ObjectWrapper
{
  struct ObjectBase
  {
    virtual int operator()() const = 0;
    virtual ~ObjectBase() {}
  };

  int operator()() const;
  std::shared_ptr<ObjectBase> wrappedObject;
};
```

type_erasure18.cpp hosted with ♡ by GitHub view raw

The child classes must be template classes so that compiler can generate them.

```
struct ObjectWrapper
{
  struct ObjectBase
  {
```

```
    virtual int operator()() const = 0;
    virtual ~ObjectBase() {}
  };

  template<typename T>
  struct Wrapper : public ObjectBase
  {
    Wrapper(const T& t) :
      wrappedObject(t) {}

    int operator()() const override
    {
      return wrappedObject();
    }

    T wrappedObject;
  };

  int operator()() const;
  std::shared_ptr<ObjectBase> wrappedObject;
};
```

type_erasure19.cpp hosted with ♡ by GitHub view raw

Provide a constructor that accepts different types (obviously)

The final step is to create a template function for our constructor to accept different types.

```
struct ObjectWrapper
{
```

```
3     template <typename T>
4     ObjectWrapper(T&& obj) :
5       wrappedObject(std::make_shared<Wrapper<T>>(std::forward<T>(obj))) { }
6
7     struct ObjectBase
8     {
9       virtual int operator()() const = 0;
10      virtual ~ObjectBase() {}
11    };
12
13    template<typename T>
14    struct Wrapper : public ObjectBase
15    {
16      Wrapper(const T& t) :
17        wrappedObject(t) {}
18
19      int operator()() const override
20      {
21        return wrappedObject();
22      }
23      T wrappedObject;
24    };
25
26    int operator()() const
27    {
28      return (*wrappedObject)();
29    }
30
31    std::shared_ptr<ObjectBase> wrappedObject;
32  };
```

Visualizations

To help understand it better, it is good to visualize our wrapper class with UML diagrams. We can now wrap different types that support our common interface, ***int(*behavior)()***.

```
void PrintId(const ObjectWrapper& obj)
{
  std::cout << obj() << "\n";
}

int main()
{
  PrintId(ObjectWrapper(Object1()));
  PrintId(ObjectWrapper(Object2()));
  PrintId(ObjectWrapper(GetId));
  PrintId(ObjectWrapper(+[](){ return 4;}));
  PrintId(ObjectWrapper([i=1](){ return 4+i;}));
}
```

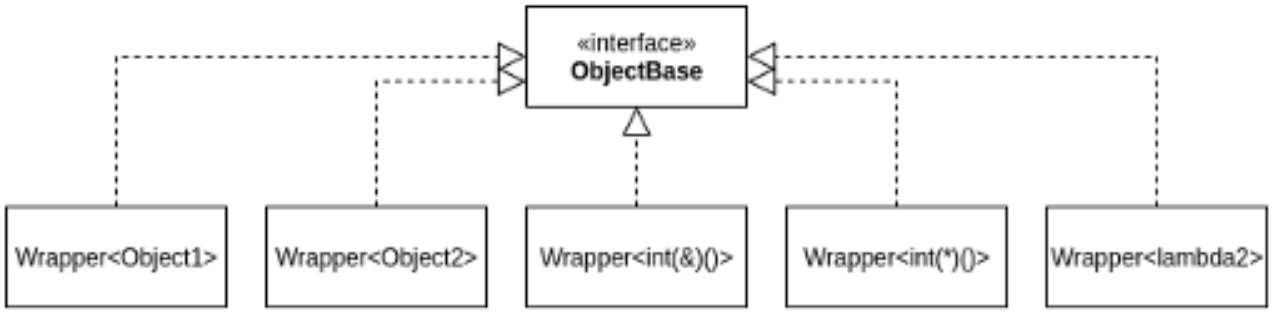
type_erasure21.cpp hosted with ♡ by GitHub view raw

We can store them in a vector as well:

```
std::vector<ObjectWrapper> objects;
objects.push_back(ObjectWrapper(Object1()));
objects.push_back(ObjectWrapper(Object2()));
objects.push_back(ObjectWrapper(GetId));
objects.push_back(ObjectWrapper(+[](){ return 4;}));
```

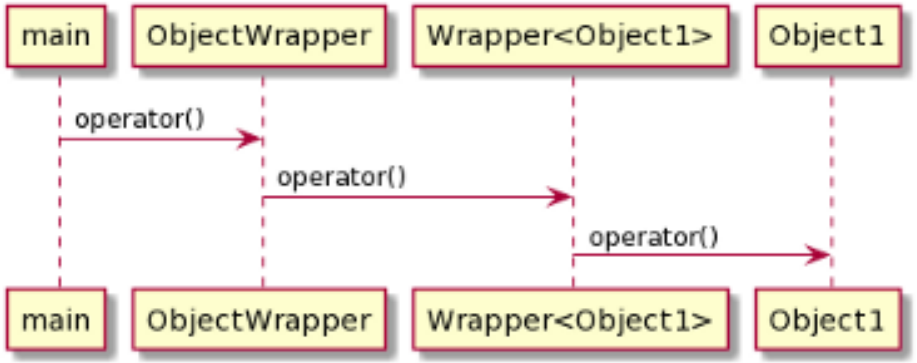
```
6   objects.push_back(ObjectWrapper([i=1](){ return 4+i;}));
```

type_erasure22.cpp hosted with ♡ by GitHub view raw



Class Diagram (Image by Author)

What happens under the hood is that we have 5 different classes that implement **ObjectBase** interface. When we invoke operator(), what happens is the following:



Sequence of events when we invoke operator() (Image by Author)

We can see that there are two extra function calls that are made, it is the cost of wrapping our objects. Another overhead at runtime that happens is the dynamic dispatch caused by our virtual functions.

Real-life Examples of this technique

Now, that we have seen the technique to wrap any type with the same behavior, we may wonder in what situations this technique is used. Two examples that you may have used frequently in your code are:

- std::function
  a general-purpose polymorphic function wrapper.
- std::any
  a type-safe container for single values of any copy constructible type.

Summary

In this post, we discuss about increasing the level of abstraction in our code by using the Type Erasure technique in C++. By combining OOP and compile-time duck typing — templates in C++, we can create wrappers that store any type.