Opaque Pointer in C++

Difficulty Level : Medium • Last Updated : 20 Jan, 2023

Opaque as the name suggests is something we can't see through. e.g. wood is opaque. An opaque pointer is a pointer that points to a data structure whose contents are not exposed at the time of its definition.

The following pointer is opaque. One can't know the data contained in STest structure by looking at the definition.

```
struct STest* pSTest;
```

It is safe to assign NULL to an opaque pointer.

```
pSTest = NULL;
```

Why Opaque pointer?

There are places where we just want to hint to the compiler that "Hey! This is some data structure that will be used by our clients. Don't worry, clients will provide its implementation while preparing the compilation unit". Such a type of design is robust when we deal with shared code. Please see the below example:

Let's say we are working on an app to deal with images. Since we are living in a world where everything is moving to the cloud and devices are very affordable to buy, we want to develop apps for windows, android, and apple platforms. So, it would be nice to have a good design that is robust, scalable, and flexible as per our requirements. We can have shared code that would be used by all platforms and then different end-point can have platform-specific code. To deal with images, we have a CImage class exposing APIs to deal with various image operations (scale, rotate, move, save, etc).

Since all the platforms will be providing the same operations, we would define this class in a header file. But the way an image is handled might differ across platforms. Like Apple can have a different mechanism to access pixels of an image than Windows does. This means that APIs might demand different sets of info to perform operations. So to work on shared code, this is what we would like to do:

Image.h: A header file to store class declaration. We will create a header file that will contain a class CImage which will provide an API to handle the image operations.

```
// This class provides API to deal with various
// image operations. Different platforms can
// implement these operations in different ways.
class CImage
{
public:
    CImage();

    ~CImage();


// Opaque pointer


    struct SImageInfo* pImageInfo;

    void Rotate(double angle);

    void Scale(double scaleFactorX,
double scaleFactorY);

    void Move(int toX, int toY);

private:
```

```
    void InitImageInfo();
};
```

Image.cpp: Code that will be shared across different endpoints. This file is used to define the constructor and destructor of CImage class. The constructor will call the InitImageInfo() method and the code inside this method will be written in accordance with the Operating System on which it will work on.

```
// Constructor for CImage

CImage::CImage() {
    InitImageInfo();
}



// destructor for CImage class

CImage::~CImage()
{
    // Destroy stuffs here
}
```

Image_windows.cpp : Code specific to Windows operating System will reside in this file.

```
struct SImageInfo {
    // Windows specific DataSet
};

void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;
    // Initialize windows specific info here
}

void CImage::Rotate()
{
    // Make use of windows specific SImageInfo
}
```

Image_apple.cpp : Code specific to Mac Operating System will reside in this file.

```
struct SImageInfo {
    // Apple specific DataSet
};
void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;

    // Initialize apple specific info here
}
void CImage::Rotate()
{
    // Make use of apple specific SImageInfo
}
```

As it can be seen from the above example **while defining the blueprint of the CImage class we are only mentioning that there is a SImageInfo data structure.**

The content of SImageInfo is unknown. Now it is the responsibility of clients(windows, apple, android) to define that data structure and use it as per their requirements. If in the future we want to develop an app for a new end-point 'X', the design is already there. We only need to define SImageInfo for end-point 'X' and use it accordingly.

Please note that the above-explained example is one way of doing this. Design is all about discussion and requirements. A good design is decided to take many factors into account. We can also have platform-specific classes like CImageWindows, and CImageApple and put all platform-specific code there.

Downsides of Opaque pointers

It's also important to note that opaque pointers can have some downsides, as well. For example, because the client code cannot access the implementation details of the object, it may be more difficult to debug or troubleshoot issues that arise. Additionally, opaque pointers can make it more difficult to understand the relationships between objects and their dependencies, which can make it harder to maintain and evolve the codebase over time.

Another potential downside is that opaque pointers can increase the complexity of the codebase, and make it harder to understand how the library is implemented. If a library is large, it can be difficult to understand the relationships between different parts of the codebase, and how they interact.

To mitigate these downsides, it's important to use opaque pointers judiciously, and to provide clear and comprehensive documentation of the library's interface. It's also

important to consider other options, such as PIMPL or Handle-Body idiom, and choose the one that best fits the needs of the project.

In summary, opaque pointers are a technique that can be used to hide the implementation details of an object and provide a level of abstraction in C++. They are useful for hiding the implementation details of an object from the client code, and also for providing a level of abstraction. However, it's important to use them judiciously and to consider other options as well.

It is also important to note that opaque pointers can cause a performance overhead, as it requires extra memory to store the pointer, and an additional level of indirection when accessing the object. It also increases the complexity of memory management as the client code cannot directly deallocate the memory of the object. The library must provide functions to handle the memory management, such as creating and destroying the object, or allocating and deallocating memory for the object.

Additionally, opaque pointers can make it more difficult to write unit tests for the client code as the actual implementation of the object is hidden from the client code. This can make it more difficult to test the client code's interaction with the object.

To mitigate these downsides, it's important to use opaque pointers judiciously, and to provide clear and comprehensive documentation of the library's interface and memory management. It's also important to consider other options, such as PIMPL or Handle-Body idiom, and choose the one that best fits the needs of the project.

In conclusion, opaque pointers are a powerful technique that can be used to hide the implementation details of an object and provide a level of abstraction in C++. However, it's important to use them judiciously and to be aware of the potential downsides, such as performance overhead, memory management, and testing. It's also important to consider other options and choose the one that best fits the needs of the project.

Questions? Keep them coming. We would love to answer.

This article is contributed by **Aashish Barnwal**. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

70

Related Articles

1. Difference between passing pointer to pointer and address of pointer to any function

2. C++ Pointer To Pointer (Double Pointer)