WIKIPEDIA

Opaque pointer

In computer programming, an **opaque pointer** is a special case of an opaque data type, a data type declared to be a pointer to a record or data structure of some unspecified type.

Opaque pointers are present in several programming languages including Ada, C, C++, D and Modula-2.

If the language is strongly typed, programs and procedures that have no other information about an opaque pointer type *T* can still declare variables, arrays, and record fields of type *T*, assign values of that type, and compare those values for equality. However, they will not be able to de-reference such a pointer, and can only change the object's content by calling some procedure that has the missing information.

Opaque pointers are a way to hide the implementation details of an interface from ordinary clients, so that the implementation may be changed without the need to recompile the modules using it. This benefits the programmer as well since a simple interface can be created, and most details can be hidden in another file.[1] This is important for providing binary code compatibility through different versions of a shared library, for example.

This technique is described in *Design Patterns* as the Bridge pattern. It is sometimes referred to as "**handle classes**",[2] the "**Pimpl idiom**" (for "pointer to implementation idiom"),[3] "**Compiler firewall idiom**",[4] "**d-pointer"** or "**Cheshire Cat**", especially among the C++ community.[2]

Contents

Examples

Ada

```
package Library_Interface is

   type Handle is limited private;

   -- Operations...

private
   type Hidden_Implementation;    -- Defined in the package body
   type Handle is access Hidden_Implementation;
end Library_Interface;
```

The type `Handle` is an opaque pointer to the real implementation, that is not defined in the specification. Note that the type is not only private (to forbid the clients from accessing the type directly, and only through the operations), but also limited (to avoid the copy of the data structure, and thus preventing dangling references).

```
package body Library_Interface is

   type Hidden_Implementation is record
      ...    -- The actual implementation can be anything
   end record;

   -- Definition of the operations...

end Library_Interface;
```

These types are sometimes called "**Taft types**"—named after Tucker Taft, the main designer of Ada 95—because they were introduced in the so-called Taft Amendment to Ada 83.[5]

C

```
/* obj.h */

struct obj;

/*
 * The compiler considers struct obj an incomplete type. Incomplete types
 * can be used in declarations.
 */

size_t obj_size(void);

void obj_setid(struct obj *, int);

int obj_getid(struct obj *);
```

```
/* obj.c */

#include "obj.h"

struct obj {
    int id;
};

/*
 * The caller will handle allocation.
 * Provide the required information only
 */

size_t obj_size(void) {
    return sizeof(struct obj);
}

void obj_setid(struct obj *o, int i) {
    o->id = i;
}

int obj_getid(struct obj *o) {
    return o->id;
}
```

This example demonstrates a way to achieve the information hiding (encapsulation) aspect of object-oriented programming using the C language. If someone wanted to change the definition of `struct obj`, it would be unnecessary to recompile any other modules in the program that use the `obj.h` header file unless the API was also changed. Note that it may be desirable for the functions to check that the passed pointer is not `NULL`, but such checks have been omitted above for brevity.

C++

```
/* PublicClass.h */

#include <memory>

class PublicClass {
 public:
  PublicClass();                                 // Constructor
  PublicClass(const PublicClass&);               // Copy constructor
  PublicClass(PublicClass&&);                    // Move constructor
  PublicClass& operator=(const PublicClass&);    // Copy assignment operator
  PublicClass& operator=(PublicClass&&);         // Move assignment operator
  ~PublicClass();                                // Destructor

  // Other operations...

 private:
  struct CheshireCat;                    // Not defined here
  std::unique_ptr<CheshireCat> d_ptr_;   // Opaque pointer
};
```

```
/* PublicClass.cpp */

#include "PublicClass.h"

struct PublicClass::CheshireCat {
  int a;
  int b;
};

PublicClass::PublicClass()
    : d_ptr_(std::make_unique<CheshireCat>()) {
  // Do nothing.
}

PublicClass::PublicClass(const PublicClass& other)
    : d_ptr_(std::make_unique<CheshireCat>(*other.d_ptr_)) {
  // Do nothing.
}

PublicClass::PublicClass(PublicClass&& other) = default;

PublicClass& PublicClass::operator=(const PublicClass &other) {
  *d_ptr_ = *other.d_ptr_;
  return *this;
}

PublicClass& PublicClass::operator=(PublicClass&&) = default;

PublicClass::~PublicClass() = default;
```

The d-pointer pattern is one of the implementations of the *opaque pointer*. It is commonly used in C++ classes due to its advantages (noted below). A d-pointer is a private data member of the class that points to an instance of a structure. This method allows class declarations to omit private data members, except for the d-pointer itself.[6] As a result,

- more of the class implementation is hidden
- adding new data members to the private structure does not affect binary compatibility
- the header file containing the class declaration only needs to include those files needed for the class interface, rather than for its implementation.

One side benefit is that compilations are faster because the header file changes less often. Note, possible disadvantage of d-pointer pattern is indirect member access through pointer (e.g., pointer to object in dynamic storage), which is sometimes slower than access to a plain, non-pointer member. The d-pointer is heavily used in the Qt[7] and KDE libraries.

See also

- Application binary interface
- Handle (computing)
- Programming idiom

References

1. Chris McKillop. "Programming Tools — Opaque Pointers" (http://community.qnx.com/sf/docman/do/downloadDocument/projects.toolchain/docman.root.articles/doc1150). QNX Software Systems. Retrieved 2019-01-16.
2. Bruce Eckel (2000). "Chapter 5: Hiding the Implementation" (http://web.mit.edu/merolish/ticpp/Chapter05.html). *Thinking in C++, Volume 1: Introduction to Standard C++* (https://archive.org/details/thinkinginc00ecke) (2nd ed.). Prentice Hall. ISBN 0-13-979809-9.
3. Vladimir Batov (2008-01-25). "Making Pimpl Easy" (http://ddj.com/cpp/205918714). *Dr. Dobb's Journal*. Retrieved 2008-05-07.
4. Herb Sutter. *The Joy of Pimpls (or, More About the Compiler-Firewall Idiom) (http://www.gotw.ca/publications/mill05.htm)*
5. Robert A. Duff (2002-07-29). "Re: What's its name again?" (http://groups.google.es/group/comp.lang.ada/msg/a886bf7922727acf). Newsgroup: comp.lang.ada (news:comp.lang.ada). Retrieved 2007-10-11.
6. *Using a d-Pointer (https://community.kde.org/Policies/Binary_Compatibility_Issues_With_C%2B%2B#Using_a_d-Pointer)* — Why and how KDE implements opaque pointers
7. "D-Pointer" (https://wiki.qt.io/D-Pointer). *Qt wiki*. Retrieved 23 Dec 2016.

External links

- The Pimpl idiom (http://c2.com/cgi/wiki?PimplIdiom)
- Compilation Firewalls (http://www.gotw.ca/gotw/024.htm)
- The Fast Pimpl Idiom (http://www.gotw.ca/gotw/028.htm)

- D-Pointers (https://community.kde.org/Policies/Binary_Compatibility_Issues_With_C%2B%2B#Using_a_d-Pointer) — KDE TechBase
- When you "XOR the pointer with a random number"[1] (http://blogs.msdn.com/michael_howard/archive/2006/01/30/520200.aspx)[2] (http://udrepper.livejournal.com/13393.html), the result is a "really opaque" pointer [3] (http://www.iecc.com/gclist/GC-faq.html#GC,%20C,%20and%20C++).
- Making Pimpl Easy (http://www.ddj.com/cpp/205918714), Vladimir Batov