Removing Implementation Details from C++ Class Declarations

Mark R. Headington
Computer Science Department
University of Wisconsin–La Crosse
La Crosse, WI 54601
headingt@csfac.uwlax.edu

Abstract

Data abstraction—a concept introduced at varying places in the CS1/CS2/CS7 sequence—separates the properties of a data type (its values and operations) from the implementation of that type. This separation of specification from implementation is achieved by encapsulating the implementation so that users of the type can neither access nor be influenced by the implementation details. Ideally, therefore, the specification should be implementation-independent.

The C++ class mechanism compromises information hiding by requiring the interface to include information—the private part of the class declaration—that is needed only for implementation purposes. This paper describes two techniques for removing details of implementation structure from the C++ class declaration and discusses the advantages and disadvantages of each.

1. Introduction

At some point in the CS1/CS2/CS7 sequence, data abstraction is introduced as a tool for managing complexity in software systems. Data abstraction, the separation of the properties of a data type from its implementation, depends heavily on *encapsulation*—the technique of minimizing interdependencies among separately-written modules by defining strict external interfaces [Mica88, Snyd86].

In object-oriented programming languages, data abstraction and encapsulation are provided by the class mechanism. The class, a program representation of an abstract data type (ADT), encapsulates both structure (the concrete *data representation* of the abstract data) and behavior (algorithms that implement operations on the abstract data). Encapsulation follows because an ADT defines only the abstract properties of the type, not the concrete data representation or the implementations of the associated operations.

To maximize the benefits of encapsulation, the ADT (hence, class) is composed of two parts: the specification and the implementation. The specification is the interface that defines, for both the user and the implementor, the services provided by the ADT without reference to implementation details. The implementation encapsulates the concrete data representation and the program code for the ADT operations. Client code cannot access encapsulated data except through the operations provided in the interface. This separation of specification from implementation yields three important benefits:

- The user does not have to know how the abstraction works in order to use it correctly.
- The implementor is guaranteed that client code cannot compromise the correctness of the implementation.
- Changes to the implementation do not require clients to be reprogrammed and recompiled.

In C++, the private part of a class declaration can—and usually does—include the declarations of the concrete data representation of an ADT:

```
#include "bool.h"    // Declarations for
                     // Boolean type

class IntStack {
public:
    Boolean IsEmpty() const;
    Boolean IsFull() const;
    void    Push( int );
    int     Top() const;
    void    Pop();
    IntStack();      // Constructor
private:
    int data[MAX_DEPTH];
    int top;
};
```

The disadvantage is that the interface represented by such a class declaration is not implementation-independent. The declaration of `IntStack` clearly discloses the data representation—an `int` array and a variable to keep track of the top of the stack. Although the reserved word `private` prevents client code from accessing the data representation directly, encapsulation and information hiding are violated in ways this paper will discuss. Sections 2 and 3 describe the problem as it affects encapsulation and information hiding, and Section 4 presents two programming techniques for strengthening the separation of specification from implementation: opaque types and abstract classes.

SIGCSE '95 3/95 Nashville, TN USA

2. Encapsulation

In C++ classes, encapsulation is enforced, in one sense, by strict access controls in the form of the `private` and `protected` reserved words. Components of the class that are `private` and `protected` are inaccessible to client code. Yet encapsulation is weakened by requiring that the private part be located in the interface. If the implementor changes the ADT's data representation, all client code must be recompiled as well as relinked. This dependency is inconsistent with the very notion of encapsulation.

Stroustrup addresses this issue in *The C++ Programming Language* [Stro91]:

> Why, you may ask, was C++ designed in such a way that recompilation of users of a class is necessary after a change to the private part? And why indeed need the private part be present in the class declaration at all? In other words, since users of a class are not allowed to access the private members, why must their declarations be present in the header files the user is supposed to read? The answer is run-time *efficiency*. On many systems, both the compilation process and the sequence of operations implementing a function call are simpler when the size of automatic objects (objects on the stack) is known at compile time.

C++ is often used for applications requiring efficient, compact code. Particularly in system programming, size and speed of programs can be critical. For many applications, however, run-time efficiency is secondary to the modularity and pliability afforded by strong encapsulation of implementation details. Section 4 of this paper describes how to remove details of the data representation from the class declaration.

3. Information Hiding

Related to encapsulation is the concept of *information hiding*—the suppression of implementation details that do not affect the *use* of an abstraction. Even though the private members of a C++ class are inaccessible to client code, they are visibly exposed to the user as a human. Here is Bertrand Meyer's perspective on information hiding [Meye88]:

> It is worth recalling ... that the purpose of information hiding is abstraction, not protection. We do not necessarily wish to prevent client programmers from accessing secret class elements, but rather to *relieve* them from having to do so. In a software project, programmers are faced with too much information, and need abstraction facilities to concentrate on the essentials. Information hiding makes this possible by separating function from implementation, and should be viewed by client programmers as help rather than hindrance.

At best, the visibility of an ADT's data representation is a distraction whereby the user may spend time speculating about further implementation details. At worst, the user may misinterpret the data representation and make decisions based on the misinterpretation. Here is a case in point.

In my CS2 course, the textbook [Head94] proceeds slowly from abstraction to implementation. Students are introduced to the properties of common data structures—lists, stacks, queues, sets—long before they learn how to implement them. For an assignment using queues as abstractions, I gave the students the specification of a character queue class (as a `.h` file) and the compiled form—but not the source code—of the implementation (`.cpp`) file. The assignment involved filling a queue and processing the characters in certain ways. This is the header file, abridged by deleting comments:

```
// Header file cqueue.h

#include "bool.h"

const int MAX_LENG = 101;

class CharQueue {
public:
    Boolean IsEmpty() const;
    Boolean IsFull() const;
    void    Enqueue( char );
    char    Front() const;
    void    Dequeue();
    CharQueue();
private:
    int  front;
    int  rear;
    char data[MAX_LENG];
};
```

The constant `MAX_LENG` is purely an implementation detail. It is needed only to tell the compiler the size of the array representing the queue.

Most students used the `IsFull()` operation to determine when the queue was full. But some students bypassed this operation, assuming (incorrectly) that the maximum queue length was 101. These students ran into trouble because the implementation algorithms use a circular buffer in which one array element must always be left empty. The maximum queue length is therefore 100, not 101. Had the private data representation and `MAX_LENG` been absent from the header file, the students would have used `IsFull()` as intended.

4. Decoupling Techniques

Given the desirability of completely decoupling an ADT's specification from its implementation, there are two techniques for removing details of the concrete data representation from a C++ class declaration. In this discussion it is assumed that the class declaration appears in a specification (.h) file and the class implementation appears in an implementation (.cpp) file.

4.1 Opaque Types

An *opaque type* is a record (struct) type whose forward declaration appears in the specification file and whose complete declaration is hidden away in the implementation file. The private data in the class declaration consists of one item only: a pointer to an object of this opaque type [Head91]. Consider the following specification of the CharQueue class:

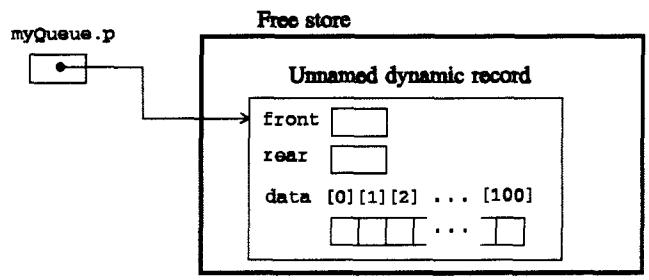
```
// Header file cqueue.h

#include "bool.h"

struct PrivateRec;// Forward declaration
                  // (Complete declara-
                  // tion is in imple-
                  // mentation file)

class CharQueue {
public:
    Boolean IsEmpty() const;
      ⋮
    void Dequeue();
    CharQueue();      // Constructor
    ~CharQueue();     // Destructor
private:
    PrivateRec* p;    // Pointer to a
};                    // PrivateRec
```

The full declaration of PrivateRec will appear in the implementation file and will describe the concrete data representation of a queue. The only private member of the class is thus a simple pointer, p.

Below is a portion of the implementation file. When a CharQueue object is created, the class constructor allocates a PrivateRec object on the free store (heap) and initializes the front and rear members to zero. When a CharQueue object is destroyed, the class destructor deallocates the private record.

```
// Implementation file cqueue.cpp

#include "cqueue.h"

const int MAX_LENG = 101;

struct PrivateRec {
    int  front;
    int  rear;
    char data[MAX_LENG];
};

CharQueue::CharQueue()    // Constructor
{
    p = new PrivateRec;
    p->front = 0;
    p->rear = 0;
}

CharQueue::~CharQueue()   // Destructor
{
    delete p;
}

void CharQueue::Enqueue( char newItem )
{
    p->rear = (p->rear + 1) % MAX_LENG;
    data[p->rear] = newItem;
}
```

If the client declares a CharQueue object named myQueue, the resulting abstract memory diagram is as follows:

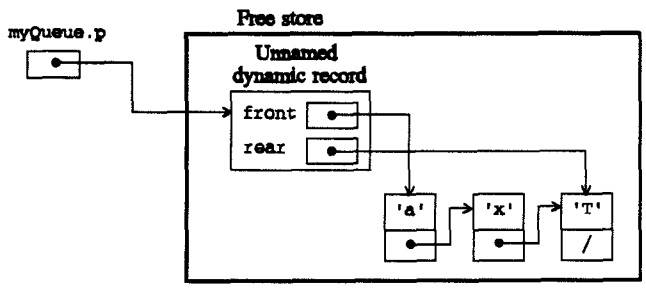


The implementor is now free to change the implementation (including the data representation) in a manner completely transparent to the client. Since the data representation is encapsulated within the implementation file rather than the specification file, the client need only be relinked, not recompiled. For example, the data representation for CharQueue might be changed to a dynamic, singly-linked list by changing the implementation file as follows:

```
// Implementation file cqueue.cpp

#include "cqueue.h"
#include <stddef.h>   // For NULL

struct NodeType {
    int       value;
    NodeType* link;
};

struct PrivateRec {
    NodeType* front;
    NodeType* rear;
};
```

```
CharQueue::CharQueue()    // Constructor
{
    p = new PrivateRec;
    p->front = NULL;
    p->rear = NULL;
}

CharQueue::~CharQueue()   // Destructor
{
     :        // Code to deallocate
              // all list nodes
    delete p;
}
```

With this implementation, the abstract memory diagram is the following:



The advantages of opaque types are twofold. First, the technique provides a total decoupling of the specification of an ADT from its implementation (not only the operations but also the data representation). This yields a purer abstraction, a stronger degree of encapsulation, and enhanced modularity. Changes to the data representation are localized in the implementation file and do not affect client code (other than the need for relinking). Second, opaque types reinforce information hiding. The user of a class can no longer be influenced by partial knowledge or misinterpretation of implementation details.

The disadvantage of opaque types is the loss of efficiency resulting from allocating and deallocating the private record on the free store as well as the extra time required for indirect addressing through pointers. This concern, along with the issue of compatibility with C language programs, led to rejection of the idea of making opaque types intrinsic to all C++ classes [Stro91]. A central theme in the design of the C++ language was that programs should not have to suffer the overhead associated with features they don't use. Since C and C++ are often used when efficiency is paramount, it is appropriate that the use of opaque types should be optional—a programming technique rather than a mandatory language mechanism.

4.2 Abstract Classes

A second technique for decoupling specification from implementation is to use an *abstract class*. In C++, an abstract class is a class with one or more *pure virtual functions*—member functions whose prototypes are prefixed with the word `virtual` and are suffixed with the symbols "`=0`". Here is the declaration of `CharQueue`, rewritten as an abstract class:

```
// Header file cqueue.h

#include "bool.h"

class CharQueue {
public:
    virtual Boolean IsEmpty() const = 0;
    virtual Boolean IsFull() const = 0;
    virtual void    Enqueue( char ) = 0;
    virtual char    Front() const = 0;
    virtual void    Dequeue() = 0;
};
```

Because there is no private part in this class declaration, no implementation details are exposed to the user (the student). But there is a catch: the compiler prevents you from creating objects of an abstract class. Using inheritance, you must derive a new class from the abstract class, adding private data and overriding the virtual functions. Then you can create objects of the derived class. For example, to use an array representation of a queue, you could derive a class `CharQueueVec` from `CharQueue` as follows. (This header file would *not* be presented to the students.)

```
// Header file cqueue2.h

#include "cqueue.h"

const int MAX_LENG = 101;

class CharQueueVec : public CharQueue {
public:
    Boolean IsEmpty() const;
    Boolean IsFull() const;
    void    Enqueue( char );
    char    Front() const;
    void    Dequeue();
    CharQueueVec();
private:
    int  front;
    int  rear;
    char data[MAX_LENG];
};
```

The corresponding implementation file `cqueue2.cpp` would look like that of the array implementation discussed earlier, apart from substitution of the identifier `CharQueueVec` for each occurrence of the identifier `CharQueue`.

Although students cannot declare objects of type `CharQueue`, they can declare variables of type `CharQueue*` (pointer to `CharQueue`) or `CharQueue&` (reference to `CharQueue`). Effectively, the student must

bundle an entire program into a function receiving a reference to a CharQueue:

```
void StudentFunc( CharQueue& myQueue )
{
    :    // Manipulate myQueue
}
```

The instructor would wrap the student's function into an "envelope"—a program that creates a CharQueueVec object and passes it to the student's function:

```
#include "queue2.h"

void StudentFunc( CharQueue& );

int main()
{
    CharQueueVec q;

    StudentFunc(q);
    return 0;
}
```

Because the member functions of the base class CharQueue are declared as virtual, run-time binding ensures that within StudentFunc, the queue operations executed are those defined by the CharQueueVec class.

Abstract classes have two benefits. First, as with opaque types, the private data representation of an ADT is hidden from the user, and changes to the data representation do not require recompiling the client code. Second, abstract classes yield better run-time performance than opaque types; dynamic data and pointers are not required in the implementations of the member functions.

However, the drawbacks to this technique in a CS1/CS2 setting are numerous:

- If the header file cqueue2.h is withheld from the student, the student cannot write a main function to test his or her code. Only the StudentFunc function can be written, to be submitted to the instructor for running.
- If the header file cqueue2.h *is* given to the student to allow a main function to be written, then information hiding is lost. The private data representation now is visible—a problem we were trying to avoid in the first place.
- Given the header file cqueue.h, the student will need some explanation of virtual functions and the "pure-specifier" (=0). This may require considerable hand-waving, especially if inheritance and run-time binding have not been discussed yet. (With opaque types, some hand-waving is also necessary to explain the pointer in the private part, but the explanation is briefer.)
- The student may ask why the abstract class CharQueue has no constructor. The answer requires discussion of the fact that constructors and destructors are not inherited by a derived class. The student then must accept on faith that the derived class written by the instructor does include a constructor.

Given these disadvantages, I find opaque types preferable to abstract classes. For additional discussion of abstract classes, see [Stro91].

5. Conclusion

The C++ class mechanism provides only partial separation of specification from implementation. For efficient translation and execution, the compiler requires the declarations of private data to appear in the class declaration.

When giving prewritten classes to students, a purer abstraction is obtained by moving an ADT's data representation from the specification (.h) file into the implementation file. This hides the data representation from the eyes of the user and allows changes to the data representation without recompiling client code.

Opaque types and abstract classes are two techniques for completely removing implementation details from the specification file. Opaque types are easier to use and explain, but they are less efficient in their implementation. However, this loss of efficiency is likely to be inconsequential in the early coursework of the computer science curriculum.

References

[Head91] M. Headington, "Opaque types in C++," *SCOOP-West 1991 Conference Proceedings*, March 1991, pp. 17-23.

[Head94] M. Headington and D. Riley, *Data Abstraction and Structures Using C++*, D.C. Heath and Company, 1994.

[Meye88] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[Mica88] J. Micallef, "Encapsulation, reusability and extensibility in object-oriented programming languages," *Journal of Object-Oriented Programming 1* (1), April/May 1988, pp. 12-38.

[Snyd86] A. Snyder, "Encapsulation and inheritance in object-oriented programming," *OOPSLA '86 Conference Proceedings*, September 1986, pp. 38-45.

[Stro91] B. Stroustrup, *The C++ Programming Language, 2nd Edition*, Addison-Wesley, 1991.