CSE 101
Introduction to Data Structures and Algorithms
ADTs and Modules in C

Introduction

This document introduces the concepts of Modules and ADTs, and describes how to implement them in the C programming language. If you are completely new to the C language, see the following assignments from (the now defunct) CMPS 12B which give a rapid introduction.

https://classes.soe.ucsc.edu/cmps012b/Spring19/lab3.pdf
https://classes.soe.ucsc.edu/cmps012b/Spring19/lab4.pdf
https://classes.soe.ucsc.edu/cmps012b/Spring19/lab5.pdf

Informally, an *Abstract Data Type* (ADT) is a collection of mathematical objects, together with some associated operations on those objects. When an ADT is used in a program, it is usually implemented in its own *module*. A module is a self-contained component of a program having a well-defined *interface* that details its role and relationship to the rest of the program. Why are ADTs necessary? The built-in data types provided by most programming languages are not powerful enough to capture the way we think about the higher level objects in our programs. This is why most languages have a type declaration mechanism that allows the user to create high level types as desired. Often the implementation of these high level types gets spread throughout a program, creating complexity and confusion. Errors can occur if the legal operations on the high level types are not well defined or are not consistently applied. The ADT concept was developed as a way to cope with these, and other, problems related to program complexity. See https://www.youtube.com/watch?v=qAKrMdUycb8&t=3714s for some interesting remarks by Barbara Liskov, the inventor of the ADT concept, on the early history of programming methodology.

Definition

An Abstract Data Type consists of two things:

(1) A set *S* of "mathematical structures", the elements of which are called *states*.
(2) An associated set of operations which can be applied to the states in *S*.

Each ADT *instance* or *object* has a current state that is one of the elements of the set *S*. The operations on *S* fall (roughly) into two categories. *Manipulation procedures* are operations that cause an ADT object to change its state. *Access functions* are operations that return information about the state without altering it. From time to time we will consider operations that belong both categories, or to neither. An ADT is an abstract mathematical construct existing apart from any program or computing device. On the other hand, ADTs are frequently implemented by a program module. We distinguish between the mathematical ADT, and its implementation in a programming language. A single ADT may have many different implementations, all with various advantages and disadvantages.

Example Consider an *Integer Queue*. In this case *S* is the set of all finite sequences of integers, and the associated operations are: Enqueue(), Dequeue(), getFront(), getLength(), and isEmpty(). The meanings of these operations are given below. One possible state for this ADT is (5, 1, -7, 2, -3, 4, 2). (It is recommended that the reader who is unfamiliar with elementary data structures such as queues, stacks, and lists, review sections 10.1 and 10.2 of [CLRS].)

Manipulation procedures

Enqueue()	Insert a new integer at the back of the queue
Dequeue()	Remove an integer from the front of the queue

Access functions

getFront()	Return the integer at the front of the queue
getLength()	Return the number of integers in the queue
isEmpty()	Return true if length is zero, false otherwise

Other examples of mathematical structures which could form the basis for an ADT are: sets, graphs, trees, matrices, polynomials, or finite sequences of such structures. In principle, the underlying set *S* could be anything, but typically it is a set of discrete mathematical objects of some kind.

An ADT instance is always associated with a particular history of states, brought about by the application of ADT operations. In our queue example we could have the following sequence starting with the empty state ( ):

Operation	State
	( )
Enqueue(5)	(5)
Enqueue(1)	(5, 1)
Enqueue(7)	(5, 1, 7)
Dequeue()	(1, 7)
Enqueue(3)	(1, 7, 3)
getLength()	(1, 7, 3)

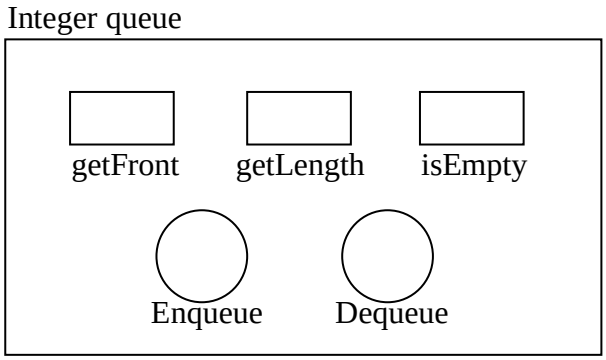
Observe that if isEmpty() is true for some state, then getFront() and Dequeue() are undefined on that state. One option to deal with this situation would be to make special definitions for Dequeue() and getFront() on an empty queue. We could for instance, define getFront() to return zero on an empty queue, and define Dequeue() to not change its state. Unfortunately, special definitions like these complicate the ADT and can easily lead to errors. A better solution is to establish *preconditions* for each operation indicating exactly when (i.e. to which states) that operation can be applied. Thus a precondition for both getFront() and Dequeue() is: "not isEmpty()". The user of an ADT must be able to determine if the preconditions for each operation are satisfied. To facilitate this, a good ADT should clearly indicate all preconditions for each operation as a sequence of access function calls. ADTs may also document their *postconditions*, i.e. conditions which *will be* true after an operation is performed. For example, a postcondition of Enqueue() is "not isEmpty()". ADT operations are in some sense analogous to mathematical functions. Preconditions and postconditions in this analogy define the function's domain and codomain. Only when all operations have been defined, along with all relevant preconditions and postconditions, can we say that an ADT has been fully specified.

Since we often work with multiple instances of an ADT, operations should specify which object is being operated on. Some ADT operations may also refer to multiple objects. For instance, the Queue ADT could have an access function called Equals() that returns true if they are in the same state, and false otherwise. We might also have a manipulation procedure called Clear() that places a Queue object in the empty state.

Some texts (including our own), define Dequeue() so as to return the front element, as well as to alter the state of the Queue, making it both an access function and a manipulation procedure. In our example,

Dequeue() deletes the front element but does not return it, making it a pure manipulation procedure rather than an operation of mixed type. Such changes in the set of ADT operations result in a *different* ADT. As a further example, suppose we implement our Queue by storing integers in an array of fixed size. We should then add an access function called isFull() that returns true if there is no room left in the array, and false otherwise. Enqueue() would then have the precondition "not isFull()". All of these variations can legitimately be called Queues, but they are to be considered different as ADTs.

It is sometimes helpful to think of an ADT object as being a 'black box' equipped with a control panel containing buttons that can be pushed (manipulation procedures), and indicator lights to be read (access functions).

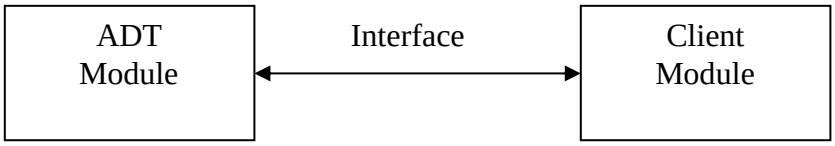


This metaphorical front panel of the black box is called the *interface* of the ADT. It is this interface that defines how an ADT will interact with other parts of a program. How the interface is implemented depends on the syntactical details the programming language.

Implementing ADTs in C

There is a straightforward way of implementing an ADT in C, once it has been specified. Many modern programming languages (like C++, Java or Python) are really made for this purpose. The C language however pre-dates the ADT concept, so implementing an ADT requires some effort.

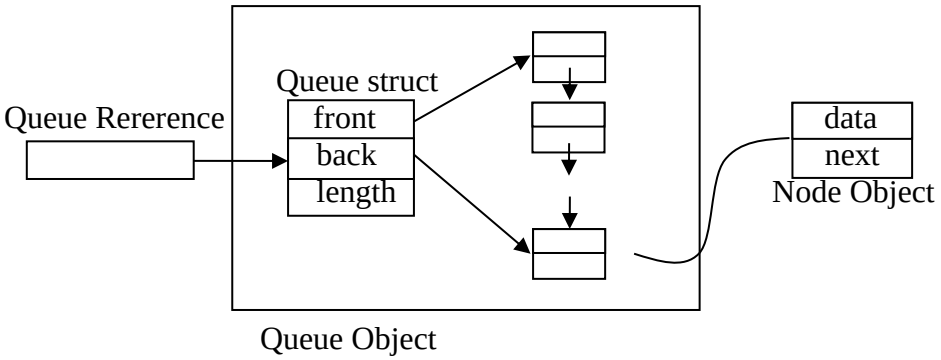
A *module* is a part of a program that is isolated from the rest of the program by a well-defined interface. We think of ADT modules as providing services (like functions or data types) to *clients*. A client is anything (program, person, computer, another module) that uses a module's services. These services are said to be *exported to* the client or *imported from* the ADT module.



The module concept includes the idea of *information hiding*: clients have no access to the internal state of a module, or any of its implementation details (i.e. what is inside the black box). The client can access a module's services only through its interface. The purpose of this principle is to reduce the complexity of the client's task by freeing it from the responsibility of knowing *how* an operation is performed. The client only knows that a given operation *will be* performed. To a client of the Queue ADT, a Queue object is just a sequence of integers that can be manipulated in certain ways. This modularity principle is actually a basic concept in all engineering disciplines. No comercially available automobile requires the driver to know how its engine works in order to drive it. A programmer building an ADT module should think in the same way.

An ADT implementation in C should contain a struct defining its "mathematical structure". The client is then given a pointer to this struct. We define one C function for each of the ADT operations, and each such function takes this pointer as an input argument. The pointer type is defined in a way that prevents the client from following the pointer to access the interior of the "black box", thus enforcing the information hiding principle.

In our Queue example, we choose the underlying data structure to be a singly linked list. (Other choices are possible, such as an array.) Thus our C implementation will contain a private struct defining a Node object. The whole thing can be pictured in memory as follows.



Two more C functions are necessary. One to create new objects (constructor) and one to free memory associated with ADT instances no longer in use (destructor). It is the responsibility of these functions to manage all of the memory inside the "black box", balancing calls to malloc(), calloc() and free().

The Queue ADT module is split into two files. A `.h` file containing typedefs and prototypes of exported functions, and a `.c` file containing struct and function definitions. The module interface consists of exactly that which appears in the `.h` file. Functions and types in the `.c` file whose prototypes do not appear in the `.h` file, cannot be accessed from outside the module, and are therefore effectively private.

```
// Queue.h
typedef struct QueueObj* Queue;

// Constructor-Destructor
Queue newQueue(void);
void freeQueue(Queue* pQ);

// Access functions
int getFront(Queue Q);
int getLength(Queue Q);
int isEmpty(Queue Q);

// Manipulation procedures
void Enqueue(Queue Q, int data);
void Dequeue(Queue Q);

// Other functions
void printQueue(Queue Q, FILE* out);
```

The file Queue.h defines a pointer called Queue, to a struct called QueueObj, which is not defined in this file. The client module will `#include` Queue.h, so the compiler will recognize calls to the exported functions. The client can also declare variables of type Queue and define functions that take Queue arguments. Notice however that the client cannot dereference a Queue variable, since the struct to which it points is not defined in Queue.h. This is how data hiding is accomplished in C. The definition of QueueObj appears in Queue.c.

```
// Queue.c
#include<stdio.h>
#include<stdlib.h>
#include "Queue.h"

// private struct, not exported
typedef struct NodeObj{
   int data;
   struct NodeObj* next;
} NodeObj;

// private reference type, not exported
typedef NodeObj* Node;

Node newNode(int data) {...}   // fill in
void freeNode(Node* pN) {...}  // fill in

// Private QueueObj struct, constructor-destructor
typedef struct QueueObj{
   Node front;
   Node back;
   int length;
} QueueObj;

Queue newQueue(void){
   Queue Q = malloc(sizeof(QueueObj));
   Q->front = Q->back = NULL;
   Q->length = 0;
   return(Q);
}

void freeQueue(Queue* pQ) {...}   // fill in

// Access functions
int getFront(Queue Q) {...}       // fill in
int getLength(Queue Q) {...}      // fill in
int isEmpty(Queue Q) {...}        // fill in

// Manipulation procedures
void Enqueue(Queue Q, int data) {...}  // fill in
void Dequeue(Queue Q) {...}   // fill in

// Other functions
void printQueue(Queue Q, FILE* out) {...}  // fill in
```

Notice that the types `NodeObj` and `Node` as well as functions `newNode()` and `freeNode()`, do not appear in the file Queue.h, and are therefore not available to the client. Exporting these items would give the client access to the inside of the black box, violating the data hiding principle. Notice also that another public function called `printQueue()` is included in both Queue.h and Queue.c. This function prints the state of a Queue object to a FILE handle (which may be `stdout`.)

We have so far left aside the question of what to do if the client calls an ADT operation in such a way as to violate one of its preconditions. We adopt the following policy in all ADT implementations.

(1) All ADT operations must state their preconditions in a comment block which appears both before the function prototype in the `.h` file, and before the function definition in the `.c` file.

(2) All ADT operations must check that those preconditions are satisfied before proceeding with nominal execution.

(3) If a precondition is violated, the ADT operation will cause the program to terminate with an error message giving: the name of the ADT module, the name of the ADT operation, and the particular precondition that was violated.

An ADT implementation should be fully tested in isolation before it is used in a larger program. The following program serves this purpose.

```
// QueueTest.c
#include<stdio.h>
#include<stdlib.h>
#include "Queue.h"

int main(int argc, char* argv[]){
   // Call each of the above functions at least once, exercising
   // every possible logical pathway through the function, including
   // error states.
   return(EXIT_SUCCESS);
}
```

Exercise Complete the function definitions above by replacing empty braces `{ . . .}` where they appear with appropriate C code. A solution to this exercise is posted on the webpage. See the section "Naming Conventions for ADTs in C" below before you do this exercise.

Some may (correctly) argue that our Integer Queue in C is not really a general purpose queue at all, and we should really write a Queue of "anythings". The problem is that C's type declaration mechanism is not advanced enough to adequately deal with this issue. There are two possible solutions. The safer one is to edit your Integer Queue to be a Queue of whatever you need a Queue of. Simply changing the appropriate instances of `int` to the new type will create a ready-made Queue ADT. This change can be accomplished efficiently by defining the type `QueueElement` in the `.h` file as follows.

```
typedef int QueueElement
```

The type `QueueElement` is used to refer to the things that are stored in a Queue. This methodology lets you change the element type by editing a single line of code. (We follow this procedure in the exercise solution posted on the webpage.)

This simple fix has the drawback that if you want `int` Queues and `double` Queues in the same program, then you need two different Queue modules. A more powerful (and less safe) technique is to make `QueueElement` a generic pointer, by doing

```
typedef void* QueueElement
```

Now the Queue module can handle Queues holding any kind of pointer. The danger here is that a client might get confused and call `getFront()` or `Enqueue()` on the wrong kind of pointer. Using `void*` means that you will not find out about this problem until you run the client program and get a segmentation fault. These types of pointer errors can be very difficult to debug. Given these warnings, I would recommend the safer solution for those students who are new to C.

Naming Conventions for ADTs in C

Suppose you wish to implement an ADT in C. The particular ADT is unimportant, so let's just call it a "Blah". You should create the following files at minimum: Blah.c, Blah.h, BlahTest.c. The file Blah.c represents the 'inside' of the black box, Blah.h represents the interface, and BlahTest.c is a kind of dummy client module or test harness used to wring the bugs out of the Blah ADT.

File Blah.h will contain prototypes for all exported ADT operations. It will also contain the line.

```
typedef struct BlahObj* Blah;
```

This defines `Blah` to be a pointer to some struct called `BlahObj`. Blah.c will `#include` Blah.h, and will contain definitions of all exported functions, as well as definitions of private functions and structs as well. Blah.c will also contain the following typedef statement.

```
typedef struct BlahObj{
   // code that defines fields for the Blah ADT
} BlahObj;
```

A client module can then `#include` Blah.h giving it the ability to declare variables of type `Blah`, as well as functions that take `Blah` parameters. However, the client cannot dereference a `Blah`, since the object it points to is not defined in Blah.h. The ADT operations take `Blah` arguments, so the client does not need to (and is in fact unable to) directly access the struct these references point to. Therefore the client can interact with a Blah object only through the exported ADT operations, thereby enforcing the information hiding principle.

Blah.c also contains a constructor:

```
Blah newBlah(...){
   Blah B;
   // code that initializes B in heap memory
   return(B);
}
```

and a destructor

```
void freeBlah(Blah* pB){
   if(pB!=NULL && *pB!=NULL){
      // free all heap memory associated with *pB
      free(*pB);
      *pB = NULL;
   }
}
```

Notice that the destructor is defined in a strange way. It's argument is not a `Blah`, but a pointer to a `Blah` (i.e. a pointer to a pointer to `BlahObj`.) Therefore the destructor is called by passing the address of a `Blah` reference. Each Blah object is explicitly created and destroyed as follows.

```
Blah B = newBlah(...);
// do something with B
freeBlah(&B);
```

Function `freeBlah()` must be defined in this way (i.e. taking a pointer to a `Blah` reference rather than a simple `Blah` reference) since it is the one ADT operation that changes the `Blah` reference itself (in addition to the `BlahObj` it points to) by setting it to NULL.

Recall that all ADT operations must check their own preconditions and exit with a useful error message when one of them is violated. The error message should state the module in which the error occurred (i.e. Blah), the operation in which it occurred, and exactly which precondition was violated. The purpose of this message is to provide diagnostic assistance to whoever is programming a client of the Blah ADT. In this course that person is of course you, the student, but in a the real world, it may well be another programmer, so you must make the error message as helpful as possible.

In the C language however, each ADT operation has at least one precondition that should be checked before all others, namely that its main reference argument is not NULL. This check must come first since any attempt to dereference a NULL pointer will result in a segmentation fault.

```
void some_op(Blah B){
   if(B==NULL){
      printf("Blah Error: calling some_op on NULL Blah reference");
      exit(EXIT_FAILURE);
   }
   // check other preconditions
   // proceed with nominal execution
}
```

In some other programming classes you may have used names like `BlahHndl`, `BlahRef` or `BlahPtr` instead of just `Blah`. We have chosen this convention in order to emulate the syntax of other OOP languages (like Python) as closely as possible. Obviously one name is not intrinsically better than another, but for the sake of consistency, you are required to adhere to the naming conventions outlined here. In particular the file names Blah.c, Blah.h, BlahTest.c; the function names `newBlah()`, `freeBlah()`, `printBlah()`; and the type names `BlahObj`, and `Blah` are not open to modification.