ADTs, Modules, and ANSI C

1 Introduction

This document introduces the concepts of Modules and ADTs, and describes how to implement them in ANSI C. An *Abstract Data Type* (or *ADT*) is a mathematical entity with an associated set of well defined operations. When an ADT is used in a program, it is usually implemented in its own *module*. Each module should be as self-contained as possible and have a well defined *interface* detailing what the module does and how it can be used.

2 Abstract Data Types

The standard or built-in types provided by many programming languages (e.g. integer, boolean, real, and character) 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 out throughout the program, creating confusion. Severe errors can be created when the legal operations on these high level types are not well defined or consistently used.

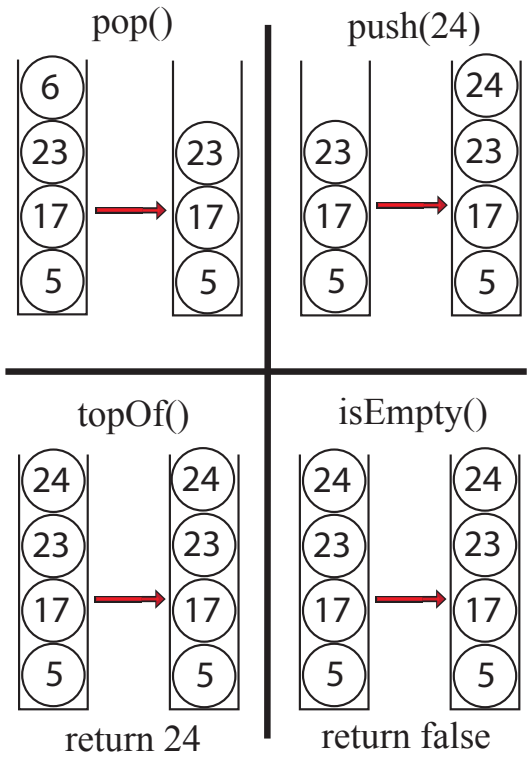
The term "Abstract Data Type" can mean different things to different people. For the purposes of this course, an abstract data type is set of mathematical structures[1] together with a group of precisely defined operations that can be applied to the structures in the set. Each ADT object has a *state* or *value* which is one of the mathematical structures in the set. Some of the operations, called *manipulation procedures*, cause the ADT object to change its state, so that its value is a different structure in the set. Other operations, called *access functions*, return information about the object without changing its state. ADTs are abstract and pure, and are defined using the language of mathematics (i.e. without any programming). On the other hand, ADTs are frequently *implemented* by a program module. We will distinguish between the mathematical ADT and the ADT's implementation in some programming language. In fact, a single ADT could have many different implementations all with various advantages and disadvantages.

Consider the simple ADT "stack of integers". The set of mathematical structures for this ADT is the set of all (finite) sequences of integers. Thus the state of a "stack of integers" at some point in time is a particular sequence of integers, with the empty sequence representing the empty stack. There are four operations that we normally associate with a stack, *push*, *pop*, *topOf*, and *isEmpty*. The manipulation procedure *push* takes a stack and an integer j. If the stack was in the state $i_1, i_2, ..., i_n$ then the push operation causes the stack to change its state to $i_1, i_2, ..., i_n, j$. The manipulation procedure *pop* is the inverse of *push*. A *pop* causes the stack to change state from $i_1, i_2, ..., i_{n-1}, i_n$ to $i_1, i_2, ..., i_{n-1}$. The access function *isEmpty* returns true if the stack's state is the empty sequence and false otherwise. The access function *topOf* returns the last integer in the sequence, so if the stack were in the state $i_1, i_2, ..., i_n$, the value i_n would be returned.

Note that both *push* and *pop manipulate* the state of the stack without returning information. while *topOf* and *isEmpty* examine the current state of the stack without making any changes. *Unlike some descriptions of the stack operations, here the pop operation does not return a value.* One of the key ideas is the separation between operations that manipulate the ADT's state and operations that

[1] Mathematical structures include objects like sequences, trees, graphs, sequences of trees, etc.

examine the ADT's state (but don't change it). You can think of an ADT object as a "black box" with buttons that can be pressed (manipulation operations) and indicators that can be read (examination operations). Good ADTs make a clear distinction between these two modes of operation.



Note that this operational definition is imprecise because the effect of a *topOf* or *pop* is not defined when the stack is empty. One option would be to define that a *pop* of an empty stack does not change the stack's state (i.e. the sequence remains empty) and that a *topOf* operation on an empty stack returns 0. Unfortunately, these special cases complicate the ADT and can easily lead to rather severe errors. A better solution is to establish *preconditions* for each of the operations indicating when the operations can be performed. The precondition for *topOf* and *pop* then becomes "not *isEmpty*". The operations *push* and *isEmpty* can be performed in any state (our mathematical sequence never "overflows"), so the preconditions for these operations are always met.

In order for an ADT to be useful, the user must be able to determine if the preconditions for each operation are satisfied. This will usually involve the use of one or more examination operations. Good ADTs clearly indicate the preconditions of each operation, usually as a sequence of examination operations. Even those operations whose preconditions are always met must have an indication to that effect.

The effects of the operations can be stated as *postconditions*. Whereas preconditions say what must be true before the operation can be performed, postconditions state what will be true after the operation is performed. Advanced methods such as axiomatic semantics provide other ways for specifying ADTs, but these advanced methods will not be dealt with here. However, the description of ADT operations should be precise enough so that the effect of *any* sequence of operations can be determined (assuming all the preconditions are met).

Note that slight changes in the stack operations lead to slightly different ADTs. If a stack is implemented with a (fixed size) array, then there is the possibility of overflow. This can dealt with at the ADT level by providing a *stackFull* access function and making "not *stackFull*" a precondition for the *push* operation. Although both this set of operations and the original ones reflect our intuition about stacks, they are different ADTs as they have different operations.

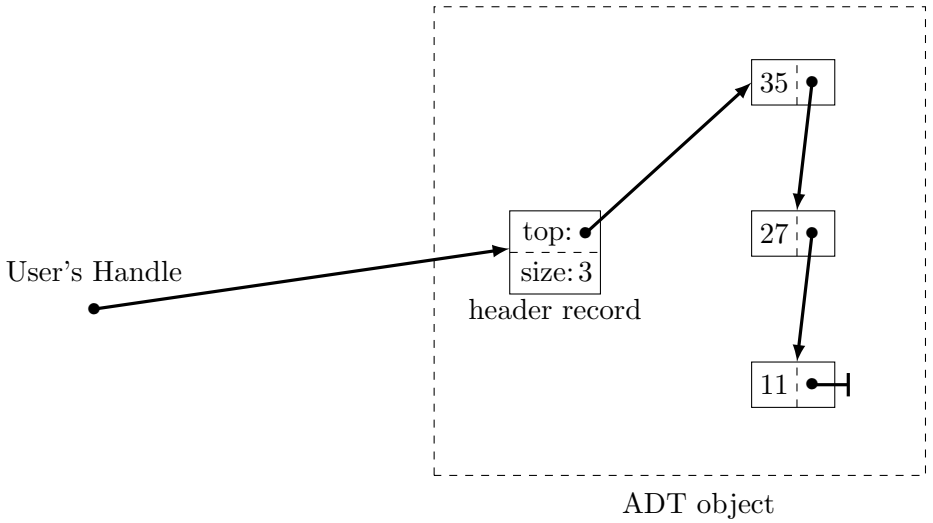
It is important to remember that each ADT is a set of mathematical structures. It is possible (and often desirable) to consider multiple objects of the same ADT. For example, we might want to talk about several stacks of integers at once. Thus the ADT operations usually will specify which stack is being operated on. It is even possible for operations to refer to multiple objects, such as an *isEqual* operation which takes two stack objects and returns true if they are representing the

same sequence, or a *concatenate* manipulation which takes two stacks, $i_1, ..., i_n$ and $j_1, ..., j_n$ and sets the first stack to $i_1, ..., i_n, j_1, ..., j_n$ and the second stack to the empty sequence.

Certain ADTs, like lists, are designed to be navigated or traversed. Operations like "Search the list for X" or "for each Y on the list do" are reasonably common things that a list ADT user should be able to do. One way of implementing these with an ADT is to have a "current position" associated with each ADT object and access functions "CurrentValue" and manipulation procedure "MoveToNext." Note that the client doesn't store the current position, but simply calls the appropriate manipulation procedure when it needs to be changed. The first assignment will have more on this point.

3 Implementing Abstract Data Types with Handles

Once an abstract data type has been defined, there is a fairly straightforward way of implementing it. Each object of the abstract type is given its own "header record," a C struct (or record in Pascal) which provides access to whatever implements the "mathematical structure". The user of the ADT gets a *handle* (usually a pointer to one of these header records) each time it creates an ADT object. These handles are used only to tell the ADT operations which ADT object is being operated on. One C function is declared for each of the ADT operations. In addition, two other C functions are generally required: one to create new objects and one which disposes of old ones. A linked list implementation of the "stack of integers" might look something like the following, when storing the sequence 11, 27, 35 (so 35 is at the "top" of the stack).



This implementation keeps a count of the number of elements on the stack. Extra information like this may be invaluable when debugging. The value of an *isEmpty* call[2] can be computed by either seeing if the `StackTop` pointer is `null` or checking if the `Size` is zero.

Note that the ADT implementation is responsible for keeping everything inside of the "Black Box" consistent. It would be a real disaster (and very difficult to debug) if the user changed

[2]In order to distinguish between the ADT operations and the functions implementing the operations, I have used *slant font* for the ADT operations and `typewriter font` for program identifiers. I also use the convention that identifiers have leading caps and when an identifier is composed of two or more words, each word starts with a capitol letter.

`StackTop` without adjusting `Size`. Therefore we insist that only the ADT implementation itself look at or change anything inside the "black box." The user manipulates the stack only by using the handle and the provided operations. Preventing the user of the ADT from interfering with the ADT implementation is extremely important and will be discussed in the section on modules.

It is easy to categorize the C functions provided by the ADT implementation into three groups.

1. C functions implementing the access functions. This group should not change the data structure associated with the ADT object. The user expects (and should have) exactly the same mathematical object before and after calling one of these functions. These functions return a value, and in this document start with a lower-case letter.

2. C functions implementing the manipulation procedures. A function which implements a manipulation procedure should not return a value, and are intended to change something about the ADT object. Here these functions start with a leading capital letter.

3. C functions which change the ADT handle. Usually there will be only two of these, the function that creates a new ADT object and the function which disposes (and reclaims the storage) of objects where are no longer needed. These also start with a leading capital.

Comments should indicate which group each function belongs to (as well as the meaning of the associated ADT operation). In addition, each of the three groups has its own style of function prototype (the function prototype is an ANSI C feature that allows one to specify what a function call looks like without giving the entire function body). For access functions use[3]

```
Boolean    isEmpty(StackHndl Stack);
int        topOf (StackHndl Stack);
```

for manipulation procedures use

```
void       Pop (StackHndl Stack);
void       Push(StackHndl Stack, int Value);
```

and for the creation/deletion procedures use

```
StackHndl       NewStack ();
void            FreeStack(StackHndl *Stack);
```

Note that ANSI C allows one to specify the type of each function argument in the function header and the return type `void` indicates that the C function is logically a procedure.

Parameters to C functions are passed by value. Thus in the group 1 and 2 calls the functions get a copy of the handle. The group 3 functions often need to change the actual handle, causing it to point to another header record or `NULL`. The `*` in the group 3 declarations indicates that the real parameter is something that points to (i.e. the address of) a stack handle, and thus the function can change the stack handle by modifying what is stored at the indicated address. This is call by reference, and is essentially how `var` parameters work in Pascal. In order to call these procedures you need to provide the address of a stack handle as in the following.

[3] "`Boolean` is not one of the C types, however a boolean type can greatly improve program readability.

```
StackHndl  MyStack;
MyStack = NewStack ();
      ...
FreeStack( & MyStack);
```

The `&` in the call causes the address of `MyStack` to be passed into the function.

Inside the `NewStack` procedure, the term `*Stack` is the stack handle; *the asterisk should be considered part of the variable's name.* Thus assigning a value to `*Stack`, such as

```
(*Stack) = NULL;
```

causes the the handle to be set to the NULL pointer. Sometimes the call by reference parameter needs to be parenthesized, if you are getting strange errors, try using `(*Stack)` instead of `*Stack`.

Another procedure, `PrintStack`, can be invaluable when you are debugging your code. This is properly a procedure and should be declared as such even though it shouldn't change the data structure representing the stack.

Allen Van Gelder has described a method of recursive ADTs where some of the manipulation procedures can change handles. Although this is a powerful technique, it can compromise safety and be error prone. I have placed his description "A Discipline of Data Abstraction using ANSI C" on the E-commons page and encourage you to read it also.

4 Modules

A module is a part of a program that is isolated from the rest of the program by a well defined interface. Modules make programs easier to write because they break up a complicated program into several much simpler pieces. Modules make programs easer to test because each module can be tested separately. Modules also make programs easier to debug since the programmer can determine which module is screwing up by watching the interface. Perhaps the biggest benefit of modules is their reusability. Once you have written a good module you can often pull it out of the old program and plug it into the new program. Finally, modules allow one to program incrementally. You can make a first pass with dumb, inefficient implementations to make sure your program design is sound. Then each module can be refined individually, perhaps by using a fancier, more efficient data structure.

Perhaps the best way to view modules is that they provide a service to *clients*. A client is anything that uses a module's services. Often, modules will use low-level services provided by other modules in order to provide some higher-level service. A service provided by a module for use in other modules is said to be *exported*. Services used by a module are said to be *imported* by that module. It is generally bad to have circular dependencies, where module A imports services from module B, module B imports services from module C, and module C imports services from module A.

One of the main ideas behind modules is *information hiding* – the clients only know the minimum amount of information necessary to correctly use a module's services and the details of the module's implementation are hidden from the clients[4].

[4]See David Parnas's articles "A Technique for Software Module Specification with Examples", *Comm. of the ACM*, **15** (5), 1972 and "On the Criteria To Be Used in Decomposing Systems into Modules", *Comm. of the ACM*, **15** (12), 1972 for more information

To accomplish this information hiding, a module is split into two parts: a *header* (in a `.h` file) indicating what services the module provides and how to use them, and an implementation of those services (in a `.c` file). The header file contains prototypes for all of the exported functions and procedures, as well as any types exported by the module. Everything used directly by a client must be exported through the header file. Of course, the effects of the functions/procedures must be also be described by comments in the header file. The key idea behind header files is that they contain all the information needed to allow someone else to use your module in their program without being cluttered with excessive implementation details. A brief description of the high level algorithm(s) used by the module is desirable, but code is not. All of the actual C code (as opposed to function prototypes and declarations of exported types) should be in the `.c` file rather than the header file.

A common misconception is that prototypes for all of a module's functions and declarations of all of the module's types must appear in the header file. *This is not true.* Only those functions which are intended to be called by a client are placed in the header file. Other internal functions can be declared (with function prototypes) in the `.c` file. Only those types that the client needs are exported through the `.h` file. Exporting too much allows the clients to see into the "Black Box" and destroys the consistency of the module's data structures.

Students often find it difficult to design good modules and module interfaces. It is just as bad to blindly put each function into its own module than it is to make large programs without using modules. Each module should be coherent in that all the services it provides are logically related. The services exported by a module should be potentially useful in another program. Modules should be used to encapsulate design decisions, so that if you change your mind, only one module needs to be changed. In general, each ADT implementation should be in its own module – for instance two files `intstack.h` and `intstack.c` might contain the interface and implementation for the stack ADT discussed earlier. However, not all modules need to be ADT implementations. For example, a sorting utility might make a good non-ADT module.

Although you may know what clients your module will have (in this program), you should treat modules as if their clients were going to be written by complete strangers. This may mean double checking some preconditions. Don't worry too much about this double checking slowing down your program. You will spend much more time debugging it than you will running it. Also, in a "production" final version (after the program has been carefully tested) these double checks can be commented out.

When implementing modules it is a *very good idea* (and required in CS 101) to write a small "driver" program that takes your newly-coded module for a "test drive" to make sure everything is working correctly. All of the operations should be called, and boundary conditions checked. You might even give the module a few error situations (like popping an empty stack) to see if its internal consistency checks are up to the job. It is far easier to debug when you are sure which module the error is in (and having just coded that particular module doesn't hurt either). It is also a good idea to re-run your driver program after any "improvements" or "fixes" to the module to check that they haven't introduced new errors. Using the driver programs in this way is called "Regression Testing".

5 Implementing Modules in ANSI C

As noted in the previous section, a module consists of a header file and a body or implementation part. The header file is the "promise" made by the module to its clients while the body (in a `.c`

file) contains the source code implementing the module.

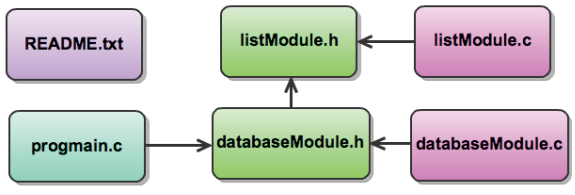
Each header file should describe (in comments) the functionality that the module exports and any other modules that are imported by this module. This includes the effect and preconditions of each exported function/procedure as well as any limitations of the implementation (such as overflows). The C statements in the module should be limited to function prototypes and type declarations. Type declarations in the `.h` file are almost always handles [5], and are usually incomplete, such as the following.

```
typedef struct IntStackStruct * StackHndl;
```

This statement declares type `StackHndl` to be a pointer to a structure called `IntStackStruct`. Although the `IntStackStruct` structure has not yet been defined (it will be defined in the `.c` file), this declaration allows the `.h` file to declare functions that take `StackHndl`'s as arguments. This declaration also allows clients to declare variables of type `StackHndl` and use them as arguments to the module's functions and procedures. Note that the client cannot reference through these pointers, that is part of the "information hiding." Only the hidden (`.c`) parts of the module should manipulate these pointers and the structures they point to. The operations exported by the module are declared using function prototypes. as indicated in the "Implementing Abstract Data Types with Handles" section. The operations imported by the module are imported through the inclusion of `.h` files *in that module's* `.c` *file*. I strongly recommend that `.h` files never include other files, and that each module's `.c` file start by including its own `.h` file and the `.h` files of the other modules that it uses. If types needed in the `.h` file are exported by other modules (i.e. defined in other `.h` files), then a big comment at the start of the `.h` file should indicate which other `.h` files are needed so any clients know what other inclusion(s) to make. Each module is responsible for allocating and freeing all of the memory used inside of its "black box." Modules may also be able to maintain their own "free lists." For example, the stack of integers example will need to allocate records for the stack elements. When a `pop` is done it may be easier to store the no-longer needed record on a list (maintained by the module) for use the next time a push is done on any integer stack. Most of you already make a directory for each program. There are a few other things that can be done to handle the many modules, header files, drivers and such. First, each directory should have a `README` file describing what the directory is for and one line which names and describes each file in the directory. Each module should have at least three files associated with it: module`.h`, module`.c`, and module`dr.c`. The module`dr.c` file contains the driver program which checks out the module. The main program should be called prog`main.c`. There should be a makefile called `Makefile` which allows the main program to be compiled with the single command "make". You will be asked to turn in all of these files for each program.

The complete code for the stack of integers module will be available on the E-commons page as an example.

Some people may (correctly) complain that the "stack of integers" is not really a general purpose stack, that writing a stack of anythings once is enough. The problem is that C's type



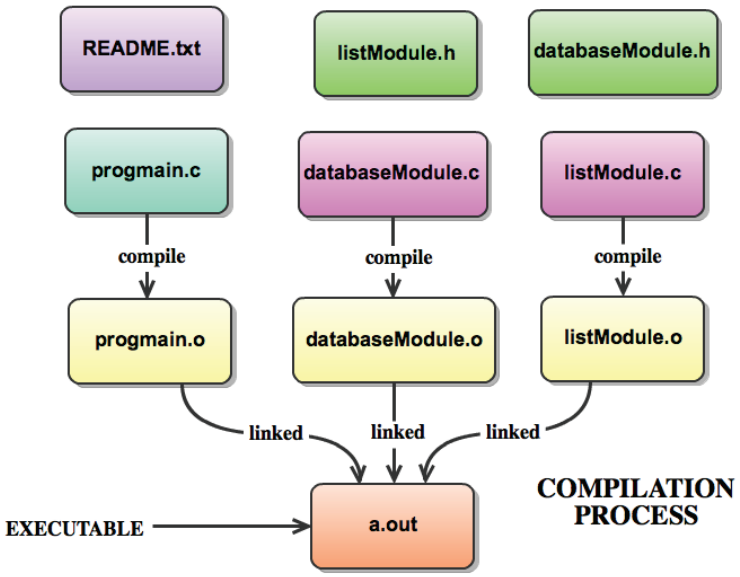
Include Structure

[5] If the client needs additional type declarations to use the module, then they should be provided in the `.h` file also, an example might be a struct type that allows a function to return two values.

mechanism is not advanced enough to properly deal with this issue. There are two possible solutions. The safer solution is to simply edit your stack of integers to be a stack of whatever it is you need a stack of. Simply by changing the appropriate `int`s to the new type will get you a ready-made stack implementation. This change can be done more easily if you use define the type `StackElType` by

```
typedef int StackElType;
```

in the `.h` file and use `StackElType` whenever you want to talk about the things that get stored on the stack. This methodology lets you change the element type by editing a single line.



These simple fixes have the drawback that if you want stacks of floats and stacks of ints in the same program, then you need two stack modules. A more powerful technique (but more dangerous) is to make the `StackElType` be `void*`, a generic pointer. Now the same stack module can handle stacks which hold any kinds of pointers.

The danger is that a client might get confused and push or pull off the wrong kind of pointer. Using `void*` means that you will not find out about this problem until you run the program and get a segmentation fault or bus error. These pointer errors can be next to impossible to debug. Although both the `typedef int StackElType;` and the generic pointer approaches are acceptable, I recommend using the safer solution in this course for those students who do not have extensive C experience.