A Discipline of Data Abstraction using ANSI C

Allen Van Gelder
University of California at Santa Cruz

Sep. 25, 2001

Contents

Abstract

This manual outlines a method for integrating data abstraction with information hiding and ANSI C. The principal intended audience is upper division undergraduate and graduate students. The main tools are header files for communication of specifications between modules, extensive use of the `typedef` facility, and function prototypes. Information hiding is achieved by separate compilation of source files.

Abstract data type (ADT) operations are divided into three types: *constructor functions* that return a reference to a new *object* of the ADT, *access functions*, which have no side effects, and *manipulation procedures*, which return no values, but may change the properties of a previously constructed object.

The method is designed for simplicity, so that it can be used by students who have never written a C program before. Although this C methodology was developed long before Java, Java was designed with the same principles in mind, and so the two are quite compatible.

1 Information Hiding

Information hiding is a technique of software development that stresses the importance of limiting the degree of module interdependence. A well-defined interface is defined for each module, and care is taken that no information inside the module is available outside unless it passes through this interface. A program that uses a module designed with information hiding is called a *client* of that module. As a software system changes and evolves, this design methodology permits clients to use a module without changing their own code as long as the module interface is not changed, even though the internals of the module (its *implementation*) may be drastically changed.

A discipline of information hiding was proposed by David Parnas[1] and has been adopted and adapted by many system developers. We shall use a variant of this discipline, somewhat simplified from the original proposal.

The interface to a module is a set of functions and procedures. Terminology is not standard across languages, but we shall adopt the Pascal notion that a *function* returns a value, while a *procedure* does not. In ANSI C and Java, a function that returns type `void` serves as our "procedure".

While it would be possible to make a data structure (`record` in Pascal, `struct` in C) part of a module interface, this tends to be counterproductive when we are seeking data abstraction, so we reject this option. Instead, the interface declares a *reference* type for objects in the ADT, and omits the layout and fields of the object. (It is noteworthy that Java has only reference types for objects; the instance fields of the object are not part of the interface, by default.) However, a *constant* might properly be part of the interface (a constant can be thought of as a function that always returns the same value).

The important feature of the module interface is that it must be *precisely describable*. Since users of the module are expected not to "peek inside", it is necessary to write down the *specifications* of its interface. To make this job reasonable, the interface must not be too complicated. To *force* a degree of simplicity, we impose certain restrictions on the functions and procedures:

Constructor functions return a reference to a new instance, or object, of the ADT.

Access functions return information about the "state" of the module or an ADT object, but *do not change* the state.

Manipulation procedures change the state, but *do not return* any information.

The importance of this severe-looking dichotomy will not be apparent until we have looked at the task of writing specifications.

David Parnas, and later, Maarten van Emden, have suggested visualizing a module as an instrument, or "black box", as shown in Figure 1. This figure shows a *stack* module as a "black box" with two "readout" buttons, labeled *empty* and *top*. When you push one of these buttons, you can get information about the current state of the box: for *empty* the light either lights up or remains off; for *top*, the display shows a value. However, these buttons cause no

[1] "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.

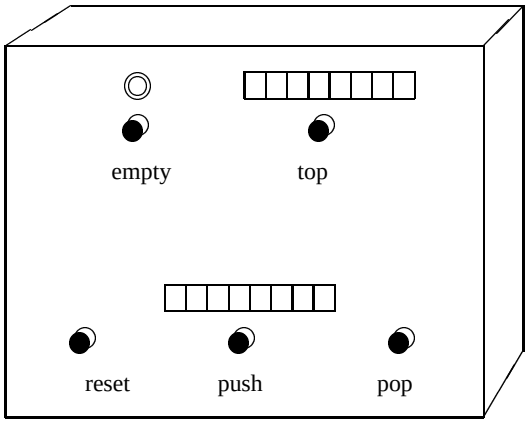

Figure 1: A "black box" with the functionality of a *stack* module.

---

change in state, so you can push them as often as you want, and still see the same responses (as long as no "transition" buttons, described next, are pushed).

The other buttons do cause state transitions, so we shall call them "transition" buttons. They work as follows:

- *reset* puts the box in a state such that the light will light up if the *empty* button is pushed; if the *top* button is pushed, the readout is unpredictable.
- To use the *push* button, first key in a value on the input keypad, then push the button. After doing so:
  1. If the *empty* button is pushed, the light remains off.
  2. If the *top* button is pushed, the readout displays the value just keyed in.
- It is proper to use the *pop* button only when the *empty* light remains off when its button is pushed. In this case, the box changes state in a way that effectively cancels the most recent uncancelled *push*. After using *pop*, the *empty* light may light, and the *top* display may be expected to change. Additional details are omitted here, but see Section 5.

2 Abstract Data Types

An *abstract data type* (ADT) is an abstract set together with a collection of operations on elements of that set. By "abstract set" we mean that the elements of the set are some conceptual entity, rather than specific storage structures. Some examples of abstract sets are

- the set of all lists
- the set of all binary trees
- the set of all graphs

For example, we can say what we mean by concatenating two lists at the abstract level without referring to their implementation in a particular program. Because most data structure algorithms are best learned at a conceptual level, the ability to be abstract without being vague is important.

Information hiding fits very well with data abstraction, for it permits us to hide the implementation details and still carry on a precise discussion. But for that discussion to be precise, we need a way to precisely describe the operations on an abstract data type without referring to unimportant implementation details.

2.1 ADT Interfaces

An *Abstract Data Type Interface* consists of a *type name*, a *set of operations* related to that type, and *specification statements* for those operations. The actual code to implement those operations is not included.

The set of operations is *complete* if they permit us to do everything we "want" to do with objects of that type. This is a necessarily vague notion, but usually makes sense. For example, if our type is "list", we should be able to construct any object we think of as a list using the available operations, and we should be able to find out any "list-related" information about any object that has been constructed without knowing the history of how it was constructed, and *without modifying the object*. When designing an abstract data type, it is worth thinking about whether a complete set of operations has been provided. Gaps often show up when attempting to write the specifications.

In our methodology ADT Interfaces are encoded in *header files* (often called "dot h files" because of the C convention of ending their names with "`.h`"). These files represent the interface between the module user and the module implementer. This is discussed in detail later.

2.2 Reference Types and Handles

Suppose we have a variable whose type is an ADT type name, say `List`. What does the value of that variable represent? Conceptually, it refers to *all* of the storage required for one object of the type. Thus type `List` is best thought of by the client as a *reference* to a complete list, *not* merely a pointer to its first "node". In fact, as discussed in Section 6, a client programmer can do everything he or she needs to do with a list without being aware of the concept of "list node", and even without knowing whether type `List` is a pointer. (That type might well be a *cursor*, that is, an integer index into some global array.) For this reason we prefer to call the type a *handle* for the list, rather than a pointer or cursor.

The person that *implements* the abstract data type naturally knows about list nodes (if they exist) and pointers or cursors. But this knowledge should remain *inside* the module. The client "sees" only what is exported from the module.

3 Some C Syntax Issues

Because C was designed as a general purpose language, it does little to dictate any particular programming paradigm. If we want to use the abstract data type paradigm, we need to do a fair amount of the work ourselves. (Java, having followed some 20 years later, pushes the programmer toward the ADT paradigm.) This section digresses into C syntax issues that we will need when we return to ADT issues in later sections.

3.1 Type Declaration in C

To avoid unnecessary complexities in your C code, type declaration should follow a limited pattern. The techniques of this section are recommended whether you are programming ADTs or not.

The approach we advocate uses `typedef`, although technically this statement is optional in C. Types should normally be defined at the beginning of a C file, right after any constants. They precede any functions or procedures, so are available for all functions and procedures *in this file* to use. To make a type available to another file as well, put it in the header file, and include the header file in all files that need that type. (See later examples.)

The **first rule** is to declare pointer types *before* describing what they point to, which is usually a `struct`. After pointer types are declared, the details of the objects to which they point may be declared. This technique, which seems backwards at first, makes it easy to declare recursive, or interleaved, data types without facing a "chicken and egg" problem.

The **second rule** is to write enough `typedefs` so that every variable can be declared by a named type in the same simple format as `ints` are declared. Normally, separate types are not declared for *arrays*, however.

If the pointer type is the type name of the ADT, it will be declared in the header file for the ADT interface. However, the `struct` type to which it points is *not* defined in that header file. Instead, the actual `struct` is defined in the C file that implements the ADT.

```
typedef struct TetraStruct * Tetrahedron;
typedef struct Point3DStruct
               {
               double  x, y, z;
               }
               Point3DStr;
typedef struct TetraStruct
               {
               double      volume;
               Point3DStr  vertex[4];
               }
               TetraStr;
```

```
void someProcedure(TetraStr oldTs[], int oldN)
    {
    Tetrahedron     curTetra;
    int             newNum
    TetraStr        newTs[20];
    TetraStr        workT;
    ...
    }
```

Figure 2: Illustration of type declarations of `structs` and pointers.

However, it may be desirable that certain *access functions* return a struct type, because they need to return more than one value. But this *return type* is different from the type of the ADT struct. In this case, the complete definition of that type is needed in the interface, and a pointer to that type is normally *not* needed.

Example 3.1: Figure 2 shows a miniature example, for 3D geometry. Because our purpose here is just to illustrate the mechanics of using `typedef` and declaring pointer types, let us assume that we are *not* defining an ADT type. Consequently, everything goes in the one C file that knows about this type. (Readers unfamiliar with C syntax should consult an ANSI C text.)

The first `typedef` declares the type `Tetrahedron` to be a pointer to a `struct` named `TetraStruct`. (In official C terminology, the name of a struct is called a *struct tag*.) The name that appears after "`struct`" is *not* a type name, which makes things very confusing at first. However, after you get used to it, it is only *mildly* confusing.

The second `typedef` defines the type `Point3DStr` to be a struct named `Point3DStruct`. The third `typedef` defines the type `TetraStr` to be a `struct` named `TetraStruct`, and also declares the fields of that `struct`, one of which is an array of 4 `Point3DStr`'s. (*See the remark at the end of this subsection about making these two names the same.*)

Now, let's check how `someProcedure` uses these declarations. In the parameter list `oldTs` is declared to be an array of `TetraStr` of unspecified length. Although the C compiler would treat this declaration the same as "Tetrahedron oldTs", the form we used tells us to expect an array, not just one object. (Probably, `oldN` is expected to tell us how many objects are in the array.) The storage space for `oldTs` has to be allocated by the caller.

Inside `someProcedure` we define our own array, `newTs`, so we have to tell the compiler how many objects are in the array. We also have defined a pointer `curTetra` to be of type `Tetrahedron`. The last line defines a variable of type `TetraStr`. □

Notice that each variable is defined in the simple format that we use for built-in types and arrays of built-in types. The complicated definitions that C allows are avoided.[2]

Experienced C programmers will probably jump on the fact that the `Tetrahedron` and `TetraStr` declarations could have been accomplished with one `typedef`. But as we shall see later, when dealing with ADTs, the two `typedef`s belong in different files.

Unfortunately, C tempts you to take shortcuts that degrade your program "structure". Resist the temptation, and your program development will follow a smoother course.

[2] A `typedef` that defines a *function* type is possible, but not shown, and is rarely needed; it would be used to define variables whose values are actual functions.

One more remark is in order here. We used two names, `TetraStruct` as the name of the `struct`, and `TetraStr` as the name of the `type`. This was to illustrate the different roles these two names play as far as C syntax is concerned. However, in practice, there is no reason not to use the same name for both, so our further examples are based on both names being the first type name, which in this example was `TetraStruct`.

3.2 Generic Pointers

ANSI C provides a way to declare that a variable is a pointer without defining what type of object it points to. Its type is called *pointer to void*, or *void pointer*. This is occasionally useful in the design of an abstract data type that represents a conceptual data structure whose element type is not important, in other words, a "generic" ADT. Because void pointers have been found to be quite error-prone we recommend against their use in most situations. The technique is described briefly for completeness, and because some C library functions use void pointers. The form of declaration might be:

```
typedef void * EltType;
...
EltType e1, e2;
```

Only limited operations are possible with generic pointers, such as `e1` and `e2` above. In fact only two operations really make sense:

1. Assignment: `e2 = e1;`
2. Comparison for equality: `e1 == e2;`

Such variables may also be passed as function parameters and returned as function values. The type `void*` is C's closest analog to the Java class `Object`.

The value of generic pointers in data abstraction is that the structural properties of the abstract data type are sometimes quite independent of the *information* or *data* it contains in the form of elements. By making the element type a generic pointer, one ADT can be used in many applications.

Warning: Generic pointers prevent the compiler from doing as much error checking as it can with typed pointers. Errors may be harder to debug.

Important Note: It is easy to forget that the *compiler* doesn't know what type a generic pointer points to, just because *you* have something in mind. If you get an "incomplete type" error, it means you tried to do too much with a generic pointer.

This can even more confusing with a *function* that returns a generic pointer. Suppose `head()` is such a function:

```
EltType  head(List L);
        ...
        if (head(edgeList)->info > 0)
```

The compiler generates an error.

The solution is to assign the generic pointer to a local pointer variable that has the correct definite type, then do your operation with that variable, as follows.

```
EltType  head(List L);
...
EdgeRecord  edge;
        ...
        edge = head(edgeList);
        if (edge->info > 0)
```

```
/****** stack.h ******/

typedef struct StackStruct * Stack;
typedef void * StackElt;

StackElt  top(Stack thisStack);
...

/****** stack.c ******/

#define  STACKSIZE  1000

typedef struct StackStruct
                {
                int     avail;
                StackElt  elt[STACKSIZE];
                }
                StackStruct;

StackElt  top(Stack thisStack)
    {
    return thisStack->elt[ thisStack->avail - 1 ];
    }
...
```

Figure 3: Function prototype in header file and function body in C file.

3.3 Function Prototypes

ANSI C permits (and may require) function prototypes, which essentially declare the types of the return value and the parameters of the function. The format is identical to the actual function header except that the prototype is followed by a semicolon, while the header is followed by a "block", that is, a pair of curly brackets enclosing code of the function. Figure 3 shows a simple function prototype, that will appear in the header file of a stack ADT (say `stack.h`), because the function `top` is exported to clients. The `typedefs` are included for completeness. Inside the C file (say `stack.c`), the function *implementation* (the header and body, in other words) is written (again following the needed `typedef`).

Be sure *not* to put a semicolon at the end of the function header when the body follows.

It is good practice to use function prototypes, even though they are not absolutely required by all C compilers. They enable the compiler to detect many errors for you.

When a function is *not* being exported from a module, you should still use a prototype, but write it in the C file, not the header file, as shown in Example 3.2, below. But in this case, it is good practice to precede the return type with the word `static`, both in the prototype and in the function header. This prevents the name from being exported unintentionally and clashing with a name elsewhere. (The same applies to a variable that is declared outside the scope of any function, and is not to be exported. Such a variable is "visible" to all the functions *in the same file* that follow it.)

3.4 Dynamic Allocation

Java, C++, and Pascal offer `new(ptr)` as a built-in function to dynamically allocate space for an object, and set pointer `ptr` to its address. Since the compiler knows the type of the pointer, it can determine how much space to allocate. In C,

there is no such compiler service. A simple solution is to write a "new" variant for each type you need to dynamically allocate.

The `malloc` and `calloc` library functions allocate space dynamically, but you need to tell them how much to allocate. The conventional and correct way to do so is with use of the `sizeof` function, which *is* supported by the compiler. (Do "`man malloc`" and "`man calloc`" for more information.)

Example 3.2: Figure 4 shows an example, based on `Stack` having been declared in the module's header file, as in Figure 5.

Notice the function prototypes. In particular, "`void *`", read as "pointer to void", is a "generic" pointer that can be changed to any type through assignment. However, the real type to which it is changed is stated in parentheses immediately after the equals sign, to confirm the programmer's intentions. This is called a "type cast".

Also notice the use of "`static`" to declare a function visible only in this module (`newStack`). Of course, you usually would not define such a simple function if it were only used in one place, but this example shows the mechanics of defining "semi-private" functions.

It seems that we violated our rule of having a `typedef` for every type by using "`StackElt*`" to define `elt`. The reason is that we wish to think of `elt` as an *array*, not as a pointer. (Java allows an array to be defined explicitly with "`StackElt[] elt`", but C does not.) For example, `thisStack->elt[thisStack->avail]` accesses the particular entry in the `elt` array where a new element would be pushed. □

The methodology of the previous example has two related drawbacks. One is that stacks must not grow larger than some predefined constant, `STACKSIZE`. The second is that space for a maximum-size stack is allocated even though it may not ever get that large. It is beyond the scope of this introductory manual to address these problems in detail; however, let us mention that another library function, `realloc`, allows us to increase or decrease the amount of space reserved by `malloc` or `calloc`, while preserving the contents of that space. If we request an increase, the entire contents may be copied to a new space. Any pointers that refer to the old space are obsolete, so great care must be taken.

Another more flexible approach appears in Example 7.1. □

4 Modules and Programs in ANSI C

The C language, including its ANSI version, is designed for development of programs whose source code is distributed among several files. Normally, all files related to a single program reside in one directory. An exception is standard files supplied as part of the system. Because we will have a variety of files related to one program, we need some standard terminology.

Definition 4.1: Files are conventionally classified as follows:

1. C file (name ends in ".c"): contains C statements for one module.
2. Object file (name ends in ".o"): contains compiled code for one module, but is not executable.
3. Header file (name ends in ".h"): contains C declarations (types, function and procedure prototypes, possibly macros and constants). A header file can be incorporated into several C files with the `#include` statement, and thus they are often called *include files*. Proper use of header files is not widely understood; it is discussed below.
4. Executable file, or program: (Name is often the name of the principal C file with the ".c" removed). It is built by linking object files together, including certain standard ones in libraries supplied by Unix. This is the name that you type to Unix to execute the program.
5. `Makefile`: The conventional name of a file that instructs the Unix `make` command how to compile and link the files in this directory into object and executable files. It can also simplify running tests and other commands.

```
/****** stack.c ******/

#include  <stdlib.h>
/* stdlib.h supplies prototypes of malloc and calloc and other functions.
 * malloc and calloc return "void *".
 */
#include  "stack.h"

#define  STACKSIZE  1000

typedef struct StackStruct
                {
                int        avail;
                StackElt*  elt;
                }
                StackStruct;

static Stack  newStack(void);
static StackElt*  newEltArray(int numElts);

static  Stack  newStack(void)
    {
    Stack   s = (Stack) malloc( sizeof(StackStruct) );
    return s;
    }

static  StackElt*  newEltArray(int numElts)
    {
    StackElt*   elt = (StackElt*) calloc(numElts, sizeof(StackElt) );
    return elt;
    }

Stack  makeEmptyStack(void)
    {
    Stack  thisStack = newStack();

    thisStack->avail = 0;
    thisStack->elt = newEltArray(STACKSIZE);
    return  thisStack;
    }
```

Figure 4: Use of `malloc()` and `calloc()` to allocate storage dynamically. Note that `malloc()` takes one parameter, the number of bytes requested, while `calloc()` takes two parameters, the number of "items" requested and the size of one "item". Also note the declaration of "`static`" functions. See Figure 5 for `stack.h`.

```
/****** stack.h ******/

typedef struct StackStruct * Stack;
typedef void * StackElt;

Stack    makeEmptyStack(int maxSize);

Boolean  emptyStack(Stack thisStack);
StackElt top(Stack thisStack);

void    push(Stack thisStack, StackElt elt);
void    pop(Stack thisStack);
```

Figure 5: Code for the header file of a `Stack` ADT. The specification comments that would be in an actual file have been omitted.

---

6. `README`: The conventional name of a file that informs the reader of the general purpose of the program(s) and test files in the directory, preferably with some examples of usage.

7. Other files, including test files, scripts, etc.

Creating too many files for one program can be just as bad as creating too few. The best course is usually to have one C file for each abstract data type and one for the "main" or "driver" procedure. Be cautious about throwing unrelated "utility" procedures together in a single file.

4.1 Header Files

Header files are one of the most misused features of the C language. The correct purpose of a header file is to specify declarations that must be *agreed upon* by two or more modules. When a module implements an abstract data type, then it should have a header file that declares the information needed to use that module, *and nothing more*.

A very common error is to throw all data structure declarations into a header file without regard to whether other modules need them.

What is the minimum necessary to be "exported" from an abstract data type module? First, the abstract data type name itself must be defined. Then the prototypes of the constructor function(s), access functions and manipulation procedures are given. In many cases, this is all the C code that should appear in the header file. However, comments should also be present here to describe the *specifications* of the abstract data type. Thus this one file is all a user (client) should need to read to successfully use the ADT; it is the "contract" between the client and the implementer.

Example 4.1: Figure 5 gives an example header file for a module that implements a stack of generic pointers. There are several points that deserve emphasis here:

1. The type defined on the first line is `Stack`, and this is the type for stack *handles*. Notice that, although it is a pointer to "`struct StackStruct`", the details of the "`struct`" itself are *not* in the header file. This is because the client *does not need to know* these details.

2. Manipulation procedure `pop` returns no value. On the other hand, the access function `top` provides access to the top of the stack without modifying it.

3. The *set* of functions and procedures in this module provide a complete interface. For example, `top` and `pop` have preconditions that the stack not be empty, but `emptyStack` provides the way to *find out* whether the preconditions hold, so erroneous calls can be avoided. (However, `emptyStack` cannot be called successfully on an uninitialized variable.) □

```
/******  client.c ******/
#include "stack.h"
typedef char * String;
...
String    x, y;
Stack     s1;
...
s1 = makeEmptyStack();
...
push(s1, y);
...
x = top(s1);
...
pop(s1);
```

Figure 6: Client code using the `Stack` ADT.

Example 4.2: Let us see the ADT method in action, from the *client's*, or *module user's* point of view. The header file (`stack.h`) contains the type name and function prototypes (Figure 5). Some fragments of client code are shown in Figure 6. □

5 Specifications

As mentioned earlier, we shall observe the discipline of defining each operation on an abstract data type as an *access function*, which returns information about objects, or a *manipulation procedure*, which modifies objects (usually one object) but returns no information. We now turn to the question of what information to provide in the specification of these operations. The answer we shall provide is probably the hardest concept to grasp in the whole topic of abstraction and modularization.

5.1 Access Function Specification

For access functions, we begin by specifying the "function prototype" for each function, which tells the type the function returns, and the types of its parameters. Then we define preconditions for each function, if any. (We do not repeat type requirements as preconditions.)

A precondition is a statement about the function's parameters that must be *and is assumed to be* satisfied when the function is called. A fundamental principle of modular design is that it is the caller's responsibility to ensure that preconditions are satisfied.[3] But to make the "game" fair, those preconditions must be verifiable. Therefore, *preconditions must be stated in terms of the other access functions*. (Functions in "lower level" modules that are also accessible to the client may also be used.) Furthermore, there must be no cycles of dependency, so the caller can actually safely call some series of functions to verify the precondition of a function without making any unsafe calls.

We assume that no *access function* can be called on an uninitialized variable; this precondition need not be stated. A variable of the ADT type must be initialized by calling a constructor function (or by assignment from another initialized variable).

So now we have specified types and preconditions for access functions. What next? For access functions, the answer is *we are done*! The return values will be entirely described as effects of the manipulation procedures.

[3] For debugging purposes it is useful for functions to take a less optimistic view of their clients and check to make sure that all arguments are valid. It is much easier to debug the error "unexpected parameter value in function foo" than "bus error – core dumped". The `assert` package is very good for this purpose; read `man assert`.

5.2 Constructor Function Specification

There is normally one *constructor function* that initializes variables, usually with a name like "create", "makeEmpty" or "makeNull" whose returned value would be stored into an uninitialized (or initialized) variable; it is often the only function that returns the type of the ADT.

For recursive ADTs, like lists and binary trees, there are at least two constructor functions, one to make a minimal object (like an empty list or empty tree), and others to make a larger object, taking smaller object(s) as parameters. However, recalling that a constant can be thought of as a function that always returns the same value, the ADT design may specify that the minimal object is some constant, which is named in the header file.

The specifications for constructor functions follow a pattern similar to that for manipulation procedures (described next), except that constructor specifications apply to a newly created object.

5.3 Manipulation Procedure Specification

The bulk of the module specification consists of describing the *effects* of the manipulation procedures and constructor functions. As with access functions, we give prototypes and preconditions, as well. The effects are often called *postconditions*. As far as possible it is desirable to describe the effects in terms of the resulting values of access functions, because these values are "observable" by the client. However, such a description, while excellent for formal specification, may be too formal and unintuitive for the average user. Therefore an informal explanation of the module is also a practical necessity.

5.4 Formal Specification

This is too big a subject to cover in depth; we give one illustrative example as a guide, using the stack.

Example 5.1: The effects of the `Stack` manipulation procedures are:

- The effect of `s = makeEmptyStack(_)` is that `emptyStack(s)` returns `TRUE`. (The underscore means that the value of this parameter is immaterial for this specification.)
- The effect of `push(s, elt)` is that `emptyStack` returns `FALSE` and that `top(s)` returns `elt`.
- The effect of `pop(s)` is to the cancel the effect of the most recent uncancelled `push(s, _)`.

To state the effect of `pop` more formally, let `s` be the stack after any correct sequence[4] of operations (manipulation procedure calls) that includes a `push` followed by a `pop`, say

$$op_1, op_2, \ldots, op_k, push(s, \_), pop(s), op_{k+3}, op_{k+4}, \ldots$$

Then `s` is "the same" after the same sequence with the `push-pop` pair removed, that is:

$$op_1, op_2, \ldots, op_k, op_{k+3}, op_{k+4}, \ldots$$

What does "the same" mean? It means all the access functions return the same values after either sequence!

Using this property that a `push-pop` pair can always be removed, we see that every stack can be "built up" with `push`'s only (following `makeEmptyStack`). □

This is a rather indirect way to describe the effect of `pop`, but with some thought we see that it really gives us all the information about stacks that we need to use them. For example, a conclusion that can be drawn is that (a correct sequence ending with) `pop` causes `emptyStack` to become true if and only if there were exactly the same number of pops as pushes after `makeEmptyStack`.

[4] A "correct sequence" is one in which preconditions are satisfied for all operations.

5.5 Specification by Pictures and Examples

A more down-to-earth way to describe effects is to give the user a conceptual picture to associate with the ADT, then show how the picture evolves with a series of "typical" operations, and how the values of the access functions evolve along with it. The art of this method is to pick a set of operations that really cover all the important cases, and allow the user to infer the important points. Drawing pictures in an ascii file can be problematical, too.

Example 5.2: Here is how an `IntStack` ADT might be described in source file comments.

```
/*
Think of ``up'' as to the right.  An IntStack is a row of ints with its
top element on the right.  An empty stack has no ints; we use a colon (:)
to show where the bottom of the stack is.

        Operation              s (after)       top     emptyStack
        ---------              ---------       ---     ----------
        s = makeEmptyStack(_)  :               undef   TRUE
        push(s, 3)             :3              3       FALSE
        pop(s)                 :               undef   TRUE
        push(s, 5)             :5              5       FALSE
        push(s, 4)             :5 4            4       FALSE
        push(s, 7)             :5 4 7          7       FALSE
        pop(s)                 :5 4            4       FALSE
        push(s, 8)             :5 4 8          8       FALSE
        pop(s)                 :5 4            4       FALSE
        pop(s)                 :5              5       FALSE

Observe that pop always cancels the effect of the most recent uncancelled push.
*/
```

□

6 Recursive Abstract Data Types

Recursive data structures are those that contain one or more fields pointing to the same type as the data structure itself. (Cycles of type references are also a possible way to create a *set* of mutually recursive types, but this is uncommon.) Two principle examples are lists and binary trees. Recursive abstract data types are the abstractions of these structures.

6.1 List ADT

Let us start by defining an abstract list recursively:

> A **list** is either an object called an*empty list*, or is an object with two parts: an *element* called its *head* and another *list*, called its *tail*.

This statement defines a list of *elements*. Normally all *elements* are of a single type, but we don't really care what that type is.

This definition differs from the nonrecursive definitions found in many texts. Nonrecursive definitions usually view the list as a sort of abstract array, with an idea of "position". Because of several drawbacks to this point of view, among them the large number of operations needed, we are adopting the methodology found in many modern languages that offer a built-in list capability, such as Lisp, Prolog, ML, and others. Instead of talking about elements at all "positions", we only mention one key position, the *head* of the list. Conceptually, the *head* of the list is the element in position 1.

Access to other "positions" is given by the recursive nature of the definition. For example, the element in position 2 of the list is simply the element in position 1 of the *tail* of the list (unless the *tail* is an empty list), i.e., the *head* of the *tail* of the original list.

Inspection of the definition suggests what *access functions* are needed for a list ADT. We can begin the definition of the `List` ADT interface with them:

```
/* list.h */

typedef struct ListNode * List;
typedef void * ListElt;

Boolean  emptyList(List list);
ListElt  head(List list);
List     tail(List list);
...
```

There are several ways to formulate a set of constructors and manipulation procedures. We shall adopt a methodology that permits the structure of a list to be modified; such structure-modifying operations are called *destructive* operations, or *mutators*, as well as similar names. For an alternate approach that uses only *nondestructive* operations (i.e., constructors only), omit `changeHead()` and `changeTail()`. Both approaches have advantages and disadvantages: *nondestructive* operations are less error-prone, while *destructive* operations may save space and run faster.[5]

```
List    makeEmptyList(void);
List    cons(ListElt newElt, List oldList);
void    changeHead(List list, ListElt newHead);
void    changeTail(List list, List newTail);
```

A convenient feature of the `List` ADT is that the specifications for the constructors can be written entirely in terms of the effects on the access functions; no hidden functions are needed, and statements about sequences of operations are unnecessary. These specifications appear in their entirety in Figure 7.

To see how these procedures work, let us trace the values of two `List` variables, `x` and `t`, through several operations. We suppose they are lists of character strings. In picturing what is happening, it is more important to form a mental picture than to worry about actual storage contents. Consequently, we picture the lists as a sequence of values between square braces.

Statement	x value	t value
x = makeEmptyList()	[ ]	
x = cons("Denver", x)	[Denver]	
x = cons("Anahime", x)	[Anahime, Denver]	
changeHead(x, "Anaheim")	[Anaheim, Denver]	
t = tail(x)	[Anaheim, Denver]	[Denver]
t = cons("Chicago", t)	[Anaheim, Denver]	[Chicago, Denver]
changeTail(x, t)	[Anaheim, Chicago, Denver]	[Chicago, Denver]

The last three statements form a common pattern for destructive operations:
(1) get the old tail of `x` into a temporary variable `t`,
(2) modify `t` as needed for the task,
(3) install the modified value of `t` as the new tail of `x` using the destructive procedure.

[5] Java and Lisp include a *garbage collector* to reclaim objects that are no longer referenced, making destructive operations by the programmer less important. C and C++ do not include a garbage collector, so in a production setting, programmers need to be sure they free all allocated space that is no longer needed, before losing track of it forever.

Figure 7: Specifications for the `List` recursive ADT.

```
------------------------------------------------------------------------
PRECONDITIONS OF ACCESS FUNCTIONS:
        emptyList(x): none.
        head(x): emptyList(x) == FALSE.
        tail(x): emptyList(x) == FALSE.
------------------------------------------------------------------------

List    makeEmptyList(void);

PRECONDITIONS: none.
POSTCONDITIONS: emptyList(makeEmptyList()) == TRUE.
        Any two calls to makeEmptyList return the same value.
------------------------------------------------------------------------

List    cons(ListElt newElt, List oldList);

PRECONDITIONS: none. (Since oldList is a parameter, it goes without
        saying that it is initialized to a value of type List.)
POSTCONDITIONS: After x = cons(newElt, oldList):
        emptyList(x) = FALSE;
        head(x) = newElt;
        tail(x) = oldList.
        The value of x is different from that of any existing List.
------------------------------------------------------------------------

void    changeHead(List x, ListElt newHead);

PRECONDITIONS: emptyList(x') = FALSE.
(Recall that x' denotes the data structure x as of when the procedure began.)

POSTCONDITIONS: emptyList(x) = FALSE; head(x) = newElt; and
tail(x) = tail(x').
(That is, this procedure leaves the value of tail unchanged.)
------------------------------------------------------------------------

void    changeTail(List x, List newTail);

PRECONDITIONS: emptyList(x') = FALSE.

POSTCONDITIONS: emptyList(x) = FALSE; tail(x) = newTail;
head(x) = head(x').
(That is, this procedure leaves the value of head unchanged.)
------------------------------------------------------------------------
```

This sequence is quite flexible by varying the details of step (2). In the above sequence, we spliced a new element into the middle of the list `x`. Continuing from the above statements, we now delete an element from the middle of a list:

```
Statement                        x value               t value
---------                        -------               -------
t = tail(x)                 [Anaheim,Chicago,Denver] [ Chicago, Denver ]
t = tail(t)                 [Anaheim,Chicago,Denver]     [ Denver ]
changeTail(x, t)                [Anaheim,Denver]          [ Denver ]
```

In this case step (2) was simply the destructive assignment, `t = tail(t)`. Observe that this statement effectively deletes the *head* element of the list `t`.

It is important to realize that destructive procedures allow us to do silly things, too. The unintended side effects that can result when two lists share elements often cause very confusing errors. Continuing from the above statements:

```
Statement                        x value               t value
---------                        -------               -------
t = makeEmptyList()            [Anaheim,Denver]          [ ]
changeTail(x, t)                  [Anaheim]              [ ]
```

All reference to the part of the list containing `Denver` (and any elements that might have been after it) has been lost.

6.2 Binary Tree ADT

The other important recursive ADT is the `BinTree` ADT, which represents binary trees. We shall discuss this ADT more briefly, as many of the issues are analogous to the `List` ADT, and recursive treatments of binary trees are easily found in good data structures texts.

A binary tree is either empty or has a root node, which has two children that are themselves binary trees (but possibly empty). The root node represents an element of type BinTreeElt. The typedefs and access functions for BinTree are

```
/****** bintree.h ******/

typedef struct BinTreeNode * BinTree;
typedef void * BinTreeElt;

Boolean    emptyTree(BinTree t);
BinTreeElt rootElt(BinTree t);
BinTree    leftChild(BinTree t);
BinTree    rightChild(BinTree t);
```

All require `t` to be initialized, and the last 3 require `emptyTree(t) == FALSE`.

The constructor procedures for BinTree are:

```
BinTree  makeEmptyTree(void);
BinTree  makeSingleton(BinTreeElt elt);
```

The manipulation procedures for BinTree are:

```
void    changeRoot(BinTree t, BinTreeElt newElt);
void    changeLCh(BinTree t, BinTree newLCh);
void    changeRCh(BinTree t, BinTree newRCh);
```

The effects of the procedures can be described in terms of the access functions, as described in Figure 8. Note that `changeLCh` and `changeRCh` are the analogs of `changeTail` in the `List` ADT.

The main difference between the way BinTrees are built up and the way Lists are built up, is that there is no idea of "attaching" a new element to an existing BinTree. Instead, `makeSingleton` creates a new BinTree with one element and two children that are empty trees.[6]

6.3 Simple Implementations for Lists and Binary Trees

Lists are so prevalent that many modern languages offer a built-in implementation, such as Lisp, Prolog, ML, and others. This section describes an implementation that is similar. The modifications for binary trees are straightforward. Remember that the material in this section is *not part of the List ADT specification*. The List client need not know it, and should not rely on it.

The struct for a `ListNode` is given, inside `list.c`, as:

```
typedef struct ListNode
        {
        ListElt  info;
        List     next;
        }
        ListNode;
```

Every *nonempty* list points to such a `ListNode`, while every *empty* list is represented by the `NULL` pointer. Thus `makeEmptyList()` becomes trivial, but `cons()` must dynamically acquire some space that will not be released when the function exits. The simplest way to do this is with `malloc()` (see Section 3.4).

```
List    makeEmptyList(void)
        {
        return NULL;
        }

List    cons(ListElt newElt, List oldList)
        {
        List    newList = malloc(sizeof(ListNode));

        newList -> info = newElt;
        newList -> next = oldList;
        return newList;
        }
```

Notice the lack of error checking. If `malloc()` cannot acquire the requested space, it will return `NULL` and the following instruction will cause an abnormal end to the program. But the List ADT specifications did not *call for* any error checking, so it is not this module's job to worry about that.

Knowing the definition of `ListNode`, the remaining List ADT operations (which do not create or destroy ListNodes) are simple, one-line functions. E.g., `head(L)` simply returns `L -> info`, etc.

The following code fragment shows how `makeSingleton()` can be simply implemented, inside `bintree.c`:

```
typedef struct BinTreeNode
        {
        BinTreeElt  info;
```

[6] But an alternative constructor might be `consBinTree(BinTreeElt newRoot, BinTree newLCh, BinTree newRCh)`, returning type `BinTree`.

Figure 8: Specifications for the `BinTree` recursive ADT.

```
-------------------------------------------------------------------------
BinTree  makeEmptyTree(void);

PRECONDITIONS: none.
POSTCONDITIONS: emptyTree(makeEmptyTree()) == TRUE.
        Any two calls to makeEmptyTree return the same value.
-------------------------------------------------------------------------

BinTree  makeSingleton(BinTreeElt elt);

PRECONDITIONS: none.
POSTCONDITIONS: After  t = makeSingleton(elt):
    emptyTree(t) == FALSE;
    rootElt(t) == elt;
    emptyTree(leftChild(t)) == emptyTree(rightChild(t)) == TRUE.
    The value of t is different from that of any existing BinTree.
-------------------------------------------------------------------------

void    changeRoot(BinTree t, BinTreeElt newElt);

PRECONDITIONS: emptyTree(t') == FALSE.
(A prime is used to represent the original value of a data structure.
 Thus t' denotes the data structure t when the procedure begins.)

POSTCONDITIONS:
    emptyTree(t) == FALSE;
    rootElt(t) == newElt;
    leftChild(t) == leftChild(t');
    rightChild(t) == rightChild(t').
(That is, this procedure leaves the children unchanged.)
-------------------------------------------------------------------------

void    changeLCh(BinTree t, BinTree newLCh);

PRECONDITIONS: emptyTree(t') == FALSE.

POSTCONDITIONS:
    emptyTree(t) == FALSE;
    leftChild(t) == newLCh;
    rootElt(t) == rootElt(t');
    rightChild(t) == rightChild(t').
(That is, this procedure leaves the rootElt and right child unchanged.)
-------------------------------------------------------------------------

void    changeRCh(BinTree t, BinTree newRCh);

PRECONDITIONS, POSTCONDITIONS: the obvious variation on changeLCh().
-------------------------------------------------------------------------
```

```
        BinTree     leftCh;
        BinTree     rightCh;
        }
        BinTreeNode;

BinTree  makeSingleton(BinTreeElt elt)
        {
        BinTree    newTree = malloc(sizeof(BinTreeNode));

        newTree -> info = elt;
        newTree -> leftCh = NULL;
        newTree -> rightCh = NULL;
        return newTree;
        }
```

Other operations are analogous to List operations.

7 Hierarchies of ADTs

It is often convenient to implement one ADT using a "lower level" ADT as a building block. Often, one purpose is to restrict the use of the lower level ADT to appropriate operations for the higher level ADT. Another purpose is to add functionality that is too specialized for the lower level ADT.

Example 7.1: The header file for a Stack ADT was shown in Figure 5. Compare the implementation in Figure 9 with the implementation suggested in Example 3.2. The `malloc` function is discussed in Section 3.4.

Observe that `thisStack->actualStack` is the "actual" handle, and may be modified by manipulation procedures. However, the value of `thisStack` itself remains as established by `makeEmptyStack`. □

This degree of complexity and indirection may be beneficial because the maximum size of a `Stack` object is not predictable when it is created. If you have a reasonable bound on the size of your ADT objects, your "constructor" procedure can simply allocate the required space and "point" the handle to it.

8 Conclusion

ANSI C has many, many features. Some of them, as explained in this manual, can be used for very effective implementation of Abstract Data Type interfaces. Others just give you an opportunity to get into trouble by being too fancy. In the "old days" some of these fancy features might have helped your program run faster, but modern compilers can usually optimize at least as well as you can.

So the principal remaining value of many features is to impress your friends with how clever you are. But remember, if the program isn't working yet, not only will your friends not be very impressed, but neither will the person grading your assignment!

So always strive for simplicity to get the first version working. Then keep a backup that represents the best working version so far, as you make "micro-improvements", that is, changes to the code that still do the same basic computation, but maybe with fewer instructions. Test whether your micro-improvements to gain speed really do (with the `time` command, e.g.); it's amazing how often they make no noticable difference, or even degrade the performance.

Improvements that make the program easier to understand are usually more worthwhile than micro-improvements to speed it up. Also, *macro*-improvements, which do an inherently more efficient *algorithm* may be worthwhile.

```
/****** stack.c ******/

#include "stack.h"
typedef struct StackStruct
        {
        List  actualStack;
        }
        StackStruct;
...
Stack  makeEmptyStack(void)
    {
    Stack  thisStack = malloc(sizeof(StackStruct));
    thisStack->actualStack = makeEmptyList();
    return thisStack;
    }

StackElt  top(Stack thisStack)
    {
    return head(thisStack->actualStack);
    }

void  pop(Stack thisStack)
    {
    thisStack->actualStack = tail(thisStack->actualStack);
    }
```

Figure 9: A Stack ADT implementation based on the List ADT as a sub-module.

Acknowledgement

The ideas in this manual were shaped by discussions with Prof. David Helmbold.

Appendix A Combining Expressions in C

C allows you to combine expressions in a statement. A typical example, where `p` and `s` are character pointers, is:

```
while (*p++ = *s++);
```

Yes, this is the *whole statement*. (See the semicolon?) There is a very simple rule that tells you when this is a good idea.

Combining expressions is never a good idea.

The reason for the above dogmatic statement can be found by rereading the earlier material of this manual about the rationale for the separation of functions (which return values, but have no effects) and procedures (which have effects, but return no values). A *statement* is inherently a procedure: no *value* is available for further use; only its *effects* persist. An *expression* is like a function: it supplies a *value* that must be used immediately or lost forever. It follows that:

An expression with side effects is bad for the same reasons as a function with side effects.

Here are some questions about what the above `while` statement does. See if you can answer them before reading on.

1. What is the minimum number of characters that might be moved?
2. Suppose `p` points to a string of length 3 (that is, 3 nonnull characters followed by null character, which is a zero) and `s` points to a string of length 5. How many characters are moved?
3. With `p` and `s` as in the previous question, by how much does the *value* of `p` change? What about the value of `s`?

You might want to jot down your answers before going to the next page.

If you had some trouble figuring out what the above `while` statement did, here is the version in which no expression has side effects.

```
*p = *s;
while (*p)
    {
    p++;
    s++;
    *p = *s;
    }
p++;
s++;
```

Consider the same questions. The correct answers are the same in both cases.

1. What is the minimum number of characters that might be moved?
2. Suppose `p` points to a string of length 3 (that is, 3 nonnull characters followed by null character, which is a zero) and `s` points to a string of length 5. How many characters are moved?
3. With `p` and `s` as in the previous question, by how much does the *value* of `p` change? What about the value of `s`?