# Abstract data types in C

TOC < PREV NEXT >

## What's an abstract data type?

You're well acquainted with data types by now, like integers, arrays, and so on. To access the data, you've used operations defined in the programming language for the data type, for instance by accessing array elements by using the square bracket notation, or by accessing scalar values merely by using the name of the corresponding variables.

This approach doesn't always work on large programs in the real world, because these programs *evolve* as a result of new requirements or constraints. A modification to a program commonly requires a change in one or more of its data structures. For instance, a new field might be added to a personnel record to keep track of more information about each individual; an array might be replaced by a linked structure to improve the program's efficiency; or a bit field might be changed in the process of moving the program to another computer. You don't want such a change to require rewriting every procedure that uses the changed structure. Thus, it is useful to *separate* the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

Here are some examples.

- stack: operations are "push an item onto the stack", "pop an item from the stack", "ask if the stack is empty"; implementation may be as array or linked list or whatever.
- queue: operations are "add to the end of the queue", "delete from the beginning of the queue", "ask if the queue is empty"; implementation may be as array or linked list or heap.
- search structure: operations are "insert an item", "ask if an item is in the structure", and "delete an item"; implementation may be as array, linked list, tree, hash table, ...

There are two views of an abstract data type in a procedural language like C. One is the view that the rest of the program needs to see: the names of the routines for operations on the data structure, and of the instances of that data type. The other is the view of how the data type and its operations are implemented. C makes it relatively simple to hide the implementation view from the rest of the program.

## Implementation of abstract data types in C

In C, a complex data type is typically represented by a pointer to the information stored in the data type. A C function can pass the pointer to other functions without knowing the details of what the pointer points to. One may also use parts of a program that have been separately compiled. All that a part of a program need know about functions it calls is their names and the types they return.

Thus, the convention in C is to prepare two files to implement an abstract data type. One (whose name ends in ".c") provides the implementation view; it contains the complete declaration for the data type, along with the code that implements its associated operations. The other (whose name

ends in ".h") provides the abstract view; it contains short declarations for functions, pointer types, and globally accessible data.

The abstract data type "stack" might then be represented as follows:

**stack.h**

```
typedef struct StackStructType *StackType;

/* Return a pointer to an empty stack. */
extern StackType InitStack ( );
/* Push value onto the stack, returning success flag. */
extern boolean Push (int k);
/* Pop value from the stack, returning success flag. */
extern boolean Pop ( );
/* Print the elements of the stack. */
extern PrintStack (StackType stack);
```

**stack.c**

```
#include "stack.h"
#define STACKSIZE 5
struct StackStructType {                /* stack is implemented as */
        int stackItems [STACKSIZE];     /*  an array of items */
        int nItems;     /*  plus how many there are */
};
typedef struct StackStructType *StackType;

/*
**      Return a pointer to an empty stack.
*/
StackType InitStack ( ) {
        char *calloc( );
        StackType stack;
        stack = (StackType) calloc (1, sizeof (struct StackStructType));
        stack->nItems = 0;
        return (stack);
}
...
```

Parts of the program that need to use stacks would then contain a line

```
#include "stack.h"
```

(Often the ".c" file also includes the corresponding ".h" file, as shown above, to insure consistency of function and type declarations.) The program would call `InitStack` to set up a stack, receiving in return a pointer of type `StackType`; it would access the stack by passing this pointer to `Push`, `Pop`, etc. Such a program would work no matter what the contents of `stack.c`, provided only that the implemented functions performed as specified in `stack.h`.

One more subtlety: It occasionally happens that a module #includes more than one `.h` file, and the second `.h` file also #includes the first. This produces compiler complaints about definitions

occurring more than once. The way to avoid these complaints is to use C's *conditional compiling* facility (described in K&R section 4.11.3) to make sure definitions only appear once to the compiler. Here is an example of its use with the stack code above.

```
#ifndef STACK           /* any suggestive variable name is fine */
#define STACK           /* define it if it's not already defined */
typedef struct StackStructType *StackType;
/* Return a pointer to an empty stack. */
extern StackType InitStack ( );
/* Push value onto the stack, returning success flag. */
extern boolean Push (int k);
/* Pop value from the stack, returning success flag. */
extern boolean Pop ( );
/* Print the elements of the stack. */
extern PrintStack (StackType stack);
#endif
```

The first time the `stack.h` file is encountered, the `STACK` variable won't (shouldn't) have been defined, so the body of the `#ifndef` is compiled. Since the body provides a definition of `STACK`, subsequent inclusions of `stack.h` will bypass the body, thereby avoiding multiple definitions of `InitStack`, `Push`, `Pop`, and `PrintStack`.

---

TOC < PREV NEXT >