Abstract Data Types in C

Abstract Data Types (ADTs) in C (1)

- C is not object-oriented, but we can still manage to inject some object-oriented principles into the design of C code.
- For example, a data structure and its operations can be packaged together into an entity called an ADT.
- There's a clean, simple interface between the ADT and the program(s) that use it.
- The lower-level implementation details of the data structure are hidden from view of the rest of the program.

Abstract Data Types (ADTs) in C (2)

- The implementation details can be changed without altering the ADT interface.
- We accomplish this by creating the ADT in three different files:
  - One to hold the type and constant definitions.
  - One to hold the prototypes of the functions in the ADT's (public) interface.
  - One to hold the implementations of the public and private functions.

Example: A Priority Queue ADT (1)

- Priority Queue: A finite collection of items in which each item has a priority. The highest priority item X in a priority queue PQ is an item such that (priority of X) >= (priority of Y) for all Y in PQ.
- Operations:
  - Initialize PQ to be empty.
  - Determine whether or not PQ is empty.
  - Determine whether or not PQ is full.
  - Insert a new item X into PQ.
  - If PQ is not empty, remove the highest priority item X from PQ.

Example: A Priority Queue ADT (2)

- Type declarations and the public interface are packaged separately from the implementation details.
- Each operation is represented by a function.
- Type declarations are put in a file PQTypes.h
- Public and private function implementations are put in a file PQImplementation.c
- Prototypes for functions in the public interface are put in a file PQInterface.h
- (Sometimes we combine the two header files into one file)

Example: A Priority Queue ADT (3)

The file PQInterface.h contains:

```
#include "PQTypes.h"
    /* defines types PQItem, PQueue */

void Initialize ( PQueue * );
int Empty ( PQueue * );
int Full ( PQueue * );
int Insert ( PQItem, PQueue * );
PQItem Remove ( PQueue * );
```

Example: A Priority Queue ADT (4)

- The statement #include "PQTypes.h" causes the type definitions to be available to this file during the compilation process.
- The statement #include "PQInterface.h" will be put in the file PQImplementation.c so that the compiler can check that prototypes in the .h file match those in the .c file.
- The contents of PQTypes.h are available to PQImplementation.c because of the statement #include "PQTypes.h" found in PQInterface.h

Multiple Definitions with #include(1)

- If the same .h file is included in several places in a program, it will cause multiple definition errors at compile time.
- To circumvent this problem, use #ifndef (if not defined), #define and #endif macros in the .h file:

```
#ifndef PQTypes_H
#define PQTypes_H
   … <type definitions belong here>...
#endif
```

Multiple Definitions with #include(2)

- The compiler keeps track of all identifier names defined in the program so far.
- The first time the compiler scans this file, it recognizes that PQTypes_H has not been defined, so it will scan all code between #ifndef and the matching #endif.
- This causes PQTypes_H and any other identifier found in the block to become defined.

Multiple Definitions with #include(3)

- The next time this file is scanned, the compiler recognizes that PQTypes_H has been defined, and it ignores all code between #ifndef and the matching #endif.
- Note: use a different, unique identifier with #ifnde in each .h file. If the identifier has already been defined elsewhere, code that should be scanned by the compiler will be ignored.
- (A good convention: use the prefix of the filename, with an _H at the end, as above)

Linked List -- list.h, listapi.h

- list.h

```
struct nodestr {
    int data;
    struct nodestr *next;
};
typedef nodestr node;
```

- listapi.h

```
#include "list.h"
node * search(node * head, int d);
node * add(node * head, int d);
void free(node * head);
```

Linked List -- listapi.c

```
#include <stdio.h>
#include "listapi.h"
/* Search for a node by its key d */
node * search(node * head, int d)
{
   for(; head != NULL; head = head->next)
         if ( head -> data == d)  return head;
   return NULL;
}
```

Linked List -- listapi.c

```
/* insert a node into list */
node * insert(node * head, int d) {
   node * loc;
   loc=search(  *p_head, d );
   if (loc != NULL) return head;  /* No need to change */
   else {
       node * newhead;
       newhead = malloc( sizeof(node) );
       newhead -> data = d;
       newhead -> next= head;
       return newhead;
   }
}
```

Linked List -- listapi.c

```
void free_list(node *head)
{
    node *p = head;
    while (p != NULL) {
        head = head ->next;
        free(p);
        p = head;
}
```

Linked List -- main.c

```
#include <stdio.h>
#include "listapi.h"
int main() {
  int i;
  node * loc, *list = NULL;
  for (i=0; i<15; i++)
      list = add(list, i);
  loc = search(list, 10);
  if( loc != NULL) printf("10 is found.\n");
  free_list(list);
}
```

Trees

```
struct s_node {
    int data;
    struct s_node * left;
    struct s_node * right;
};
typedef s_node node;
/* The following code illustrate how to expand the tree */
……
node * root;
root = (node *) malloc(sizeof(node));
root->left = (node *) malloc(sizeof(node));
root->left->left = root->left->right = NULL;
root->right = (node *) malloc(sizeof(node));
root->right->left = root->right->right = NULL;
……
```

Release All Nodes of a Tree

```
void release_tree( node ** p_root)
{
    node * root = (*p_root);
    if( root == NULL) return;
    release_tree( &(root->left) );  /* free the left subtree*/
    release_tree( &(root->right)); /* free the right subtree */
    free( root );              /* free the root node */
    *p_root = NULL;    /* this subtree has been released,
                          so notify the calling function */
    return;
}
```