Small and Reusable Object-Based Data Structures in C

Michel de Champlain, Ph.D
Chief Scientist
(mdec@DeepObjectKnowledge.com)

Embedded Systems Conference
Silicon Valley 2009
Class ESC-464

Abstracts

For decades, embedded systems projects in C have duplicated implementations of common data structures, such as linked lists, stacks, queues, and so on.

Newer programming languages, like Java and C#, are supporting extensive libraries of reusable data structures called collections. These collections are implemented using state of the art design patterns to make them more reliable, more portable, and more maintainable.

This class shows the benefit of similar C object-based collections especially designed and optimized for small footprint embedded systems.

These collections reduce the developmental effort through code reuse, but still give embedded systems developers full control over the size and flexibility of their code.

Key Takeaways

Learn by examples how to reuse practical, uniform and common collections in C and how they will help embedded software developers become more productive.

Prerequisites: Working experience in C with object-oriented principles would be helpful.

Contents in Brief

This class is divided in 5 modules:

1. Constraints in Embedded Systems
2. How a Memory Model Can Help
3. Creating Abstractions: from C to Object-Based
4. Developing concrete and abstract data structures in C
5. Developing abstract data types and objects in C
6. Conclusion

1. Constraints in Embedded Systems

This section presents a review on the evolution of object-orientation in modern languages and their weaknesses in supporting embedded systems. It exposes our motivation in this domain of applications.

At the end of this section, you will be able to:

- understand the evolution of object-orientation and the gap between abstractions in software and their support in MCUs
- better understand the importance of object-orientation and better mapping to promote portability and reusability

Despite Object-Orientation (OO)

Despite the evolution of OO and the development of advanced language features such as:

- interfaces,
- delegates,
- properties, etc.

the linga franca for embedded systems programmers continues to be the C programming language.

Successors of the C language

C has maintained its foothold over the years for four key reasons. It is:

- fast,
- compact,
- versatile, and
- well-supported.

It has become our "portable assembler" on all platforms.

Motivation

C/C++ languages actually:

- Reveal the underlying hardware details (memory mapping in particular)
- Rely on inlining (macros with the preprocessor)
- Allow explicit pointer manipulation and dereferencing

All are difficult to maintain (and to port).

2. How a Memory Model Can Help

One way to improve the size and the reuse of collections is the choice of a memory model.

In this module, we focus our attention on:

- An underlying memory model and
- Embedded virtual machine.

Memory Model

Hardware differences can be hidden based on a uniform and more portable memory model. Our model manages the data and the runtime stack of an embedded application and is part of an embedded virtual machine (VM) The VM is dedicated for small footprint embedded systems (SFES) or more specifically, for 8- or 16-bit microcontrollers with a maximum memory size of 64K bytes. It:

- Manages all memory allocations/deallocations/defragmentations, and
- Provides a independent memory manager written in ANSI C.

Memory Manager

With embedded and/or real-time systems, the memory manager generally uses an algorithm that is a hybrid of static and dynamic allocation.

Our memory manager is no different. For example, it may preallocate three partitions of fixed memory blocks in order to diminish fragmentation by routing and localizing the search of a free block to its appropriate partition.

Our memory manager is central to all dynamic (object) allocations.

3. Creating Abstractions: From C To Object-Based

This section presents the evolution of creating abstractions from C to Object-Based (OB or Modular).

At the end of this section, you will be able to:

- understand the three evolutionary stages in programming
- better understand the C limitations and OB advantages for flexibility, extensibility, and maintenance.

The Programming Evolution

Historically, modeling and programming small embedded systems has been almost impossible to tackle with an object-oriented approach.

For that reason, the majority of embedded systems programmers continue to use C as their programming language of choice.

Although C is also versatile and allows us to emulate object-based or object-oriented approach, doing so requires sophisticated programming techniques.

Hence, the motivation for developing collections in an OB way: write once for all and reuse often (especially when space and performance are important concerns).

The Programming Evolution

In this session, we examine the evolution of the three predominant paradigms of software development from the procedural to the object-based approach (with the advantages and disadvantages of each)

Three Stages

There are three stages (or paradigms):

1. procedural (global data + function) — CDS[1]
2. procedural (no global data + function) — ADS[2]
3. object-based (modular) — ADT[3]

[1] Concrete Data Structure.
[2] Abstract Data Structure.
[3] Abstract Data Type.

Procedural Example

Declaring three kinds of devices (constants) of a type named 'DeviceType':

```
typedef enum {BUTTON, KEYPAD, SENSOR} DeviceType;
```

Data structure representing a device:

```
typedef struct {
    DeviceType  type;
    // Other data members
} Device;                   // Declaring a structure of type 'Device'.

Device d;                   // Allocating a device 'd'.

d.type = BUTTON;            // Initializing its type as 'BUTTON'.
```

Procedural Example

Now, in using any device d, careful tests on device type must be performed:

```
if      (d.type == BUTTON)   value = ButtonRead(d);
else if (d.type == KEYPAD)   value = KeypadRead(d);
else if (d.type == SENSOR)   value = SensorRead(d);
```

or

```
switch (d.type) {
    case BUTTON:   value = ButtonRead(d); break;
    case KEYPAD:   value = KeypadRead(d); break;
    case SENSOR:   value = SensorRead(d); break;
}
```

Procedural Example

Therefore, the update is very prone to error:

```
typedef enum {BUTTON, KEYPAD, SENSOR, KEYBOARD} DeviceType;
...
switch (d.type) {
    case BUTTON:   value = ButtonRead(d);   break;
    case KEYPAD:   value = KeypadRead(d);   break;
    case SENSOR:   value = SensorRead(d);   break;
    case KEYBOARD: value = KeyboardRead(d); break;
}
```

Bear in mind that the above `switch` statement must be modified for each reference to `d.type`.

4. Developing Concrete and Abstract Data Structures in C

- Concrete Data Structures (CDSs)
  - What is a CDS?
  - Using a CDS (in C)
  - Consequences
- Abstract Data Structures (ADSs)

What is a Concrete Data Structure (CDS)?

- A CDS is a module with a visible (global) user-defined data structure
  - Used in several procedure-oriented (PO) languages
  - Very efficient, but non-reusable

CDS Interface (`StackCDS.h`): Stack of ints

```
const int SIZE = 10;

typedef struct {
    int items[ SIZE ];
    int sp;
} Stack;

Stack s;

void Stack_init()          { /* code imple. */ }
int  Stack_pop()           { /* code imple. */ }
void Stack_push(int item) { /* code imple. */ }
bool Stack_isEmpty()       { /* code imple. */ }
```

Using a CDS

A concrete stack has no parameters, implying direct access to the variable `s` of type `Stack`:

```
#include "StackCDS.h"

int  n;

Stack_init();
Stack_push( 10 );
if ( !Stack_isEmpty() ) {
   n = Stack_pop();      /* Good: via the interface     */
   n = s.items[ s.sp ]; /* Bad:  direct access to data */
   /* ... */
}
```

CDS Consequences (-)

- Users are bothered with unnecessary details
- Modifications of the data affect all users directly
- Is a one shot deal! (non-reusable code)

- Concrete Data Structures (CDSs)
- **Abstract Data Structures (ADSs)**
  - What is a ADS?
  - Using a ADS (in C)
  - Consequences

What is an Abstract Data Structure (ADS)?

- Like a CDS, an ADS is a module that:
  - Integrates data and procedures
  - But the data is hidden within the unit and can only be accessed via public procedures and functions
- Called abstract because
  - Only a qualified name (`module.operation`) is known
  - Its implementation is unknown

Abstact Data Structures

- Support information hiding [Parnas 72]:
  - If the implementation changes,
  - Then the interface remains unchanged
- Have a state that:
  - Can only be modified via public procedures
  - Represents the values of the internal data structure that cannot change between procedure invocations

An ADS Interface in C

- In C, an ADS can be implemented as the following:

```
/* StackADS.h - ADS for a stack of ints */

void Stack_init();
int  Stack_pop();
void Stack_push(int item);
bool Stack_isEmpty();
```

Using an ADS

The same way as CDS except that access to the internals will be flagged by the compiler:

```
#include "StackADS.h"
int n;

Stack_init();
Stack_push( 10 );
if ( !Stack_isEmpty() ) {
   n = Stack_pop();     /* Good: via the interface  */
   n = s.items[ s.sp ]; /* Bad: caught by compiler  */
   \*    ^compilation error: internal access to ADS */
}
```

ADS Benefits (+)

- Users do not need to be familiar with the implementation
- Implementation can be changed later without affecting the users
- Data is protected in the module (e.g. no global access)

ADS Consequences (-)

- Accessing the data through procedures is less efficient
- Module must be modified if a new operation (functionality) is needed to access the internal data
- Only one instance

5. Abstract Data Types and Objects in C

- Concrete Data Structures (CDSs)
- Abstract Data Structures (ADSs)
- **Abstract Data Types (ADTs)**
  - What is a ADT?
  - Using a ADT (in C)
  - Consequences

What is an Abstract Data Type (ADT)?

An ADT is a module:

- Used as a type
- That can instantiate several variables (objects) of this type
- That can be implemented in two ways:
  - As structures or
  - As pointers to structures

An ADT Interface in C

Implemented as a pointer to a structure:

```
/* StackADT.h - ADT for a stack of ints */

        struct Stack_Desc;           /* forward ref */
typedef struct Stack_Desc* Stack_t; /* opaque type */

Stack_t Stack_new   ();
void    Stack_init  (Stack_t  this);
void    Stack_push  (Stack_t  this, int item);
int     Stack_pop   (Stack_t  this);
int     Stack_top   (Stack_t  this);
void    Stack_delete(Stack_t  this);
```

Using an ADT

Based on the previous interface, users can now instantiate multiple variables of type `Stack_t` as follows:

```
Stack_t   aStack, anotherStack;

aStack        = Stack_new();
anotherStack = Stack_new();
```

Implementation of the ADT Stack

```
/* StackADT.c - implementation stack of ints */
#include "StackADT.h"

const int SIZE = 10;

typedef struct Stack_Desc {
   int   sp;
   int   items[ SIZE ];
} Stack_Desc;

Stack_t Stack_new () {
    Stack_t     s = (Stack_t) malloc(sizeof(Stack_Desc));
    Stack_init( s );
    return s;
}
```

Implementation of the ADT Stack

```
void Stack_init(Stack_t  s) {
    s->sp = -1;
}
void Stack_push(Stack_t  s, int  item) {
    s->items[++(s->sp)] = item;
}
int  Stack_pop (Stack_t s) {
    return s->items[(s->sp)--];
}
int  Stack_top (Stack_t s) {
    return s->items[s->sp];
}
void Stack_delete(Stack_t s) {
    free(s);
}
```

ADT: Benefits vs Consequences

- Benefits (+):
  - It extends the language by new data types
  - New data types can be used like any other basic type, such as char
- Consequences (-):
  - It is less efficient than ADSs (because of the additional parameter)
  - May involve dynamic allocations

ADS vs ADT: A Summary

- You must consider the use of:
  - ADTs if multiple instances of this type are really required

    Ex: stacks, queues, timers, ports, devices, etc.

  - ADSs if one instance is enough

    Ex: keypad, lcd display, real-time clock, etc.

6. Conclusion

Code reuse depends on two fundamental requisites:

- a uniform memory model that hides away the details of the underlying architecture and
- a structure which encapsulates data and its behavior.

7. Conclusion (con't)

As we move from a procedural to a object-based approach of embedded system development, the limitations of the C language become more apparent:

- no uniform memory model across different platforms is available and
- wrappers in languages other than C are required to mimic classes.

References

[1] M. de Champlain. Implementing a Reusable Memory Manager in ANSI C for Any Embedded Platform. Renesas Developer's Conference 2008, San Diego, November 2008. `www.renesasdevcon.com/schedule/courses/courseA11.html`.

[2] M. de Champlain and J.L. Houle. Benefits of a Memory Management Scheme Based on Object Lifetimes in Real-Time Operating Systems. *Proceedings of the Ninth IEEE Workshop on Real-Time Operating Systems and Software, Pittsburg, PA*, pages 22–25, 1992.

[3] M. Barr. Is C Passing?. *Embedded.com*, May 2002. `www.embedded.com/showArticle.jhtml?articleID=9900632`.

[4] J.W. Grenning. Why are you still using C?. *Embedded.com*, April 2003. `www.embedded.com/story/OEG20030421S0081`.