Small and Reusable Object-Based Data Types in C for Embedded Systems

Michel de Champlain*
Department of Computer Science and Software Engineering
Concordia University

Brian G. Patrick†
Department of Computing and Information Systems
Trent University

ESC-464
Embedded Systems Conference
Silicon Valley 2009

SUMMARY

For decades, embedded systems projects in C have duplicated implementations of common data structures, such as linked lists, stacks, queues, and so on. On the other hand, newer programming languages, like Java and C#, support 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 on how to reuse practical, uniform and common collections in C and how they will help embedded software developers become more productive.

Track: Software Engineering Practices

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

*Michel is chief scientist of DeepObjectKnowledge Inc., an object-oriented training/mentoring firm located in Montreal, Quebec, Canada. His research interests include programming languages for embedded systems, compilers, and virtual machines. He is also a regular speaker at the Embedded Systems Conference.

†Brian is associate professor of the Department of Computing and Information Systems at Trent University located in Peterborough, Ontario, Canada. His research interests include parallel job scheduling and programming languages.

Contents

1 Motivation

Despite the evolution of object-orientation and the development of advanced language features such as interfaces and properties, the *linga franca* for embedded systems programmers continues to be the C programming language. Introduced nearly four decades ago, the C language has spawned a lineage of programming languages including C++, Java, and most recently, C#. Yet, C itself has remained the language of choice for embedded systems applications, primarily for two key reasons. It is fast and it is compact.

The successors of C have extended the language, adding object-orientation in C++ and then a virtual machine in Java and C#. These features provide additional levels of abstraction which isolate, as much as possible, the programmer from the idiosyncrasies of the underlying architecture. From a software engineering point of view, these features are welcome news. From a embedded systems point of view, they are often cumbersome, bloated, and ill-equipped for lower level program development. This is somewhat ironic since Java was originally conceived as an embedded systems language. Equally ironic, C and C++ do not provide standard support for some basic embedded systems tasks. Among these tasks include memory access where embedded systems developers are often forced to incorporate language extensions offered by compiler vendors that permit all kinds of memory accesses or allocations via banks and data/code sections.

In this paper, we focus our attention on the benefits of C object-based collections especially designed and optimized for small footprint embedded systems. These collections also profit from a uniform memory model layer that facilitates porting across different platforms. By supporting code reuse, collections not only reduce memory requirements but also shorten the development time and effort for embedded systems applications.

2 Benefits of a Uniform and More Portable Memory Model

Although memory access is a typical embedded systems requirement, it is not standard in programming languages like C, C++, Embedded C++, and Java. In these languages, conditional compilations or special keywords are often used to support several microcontroller families. This makes applications hard to maintain

and harder still to port. Our collections submerge hardware differences based on a uniform memory model. The end result is more reusable code which is platform-independent.

One way to improve the size and the reuse of collections is the choice of a 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 as part of many embedded virtual machines (VMs). These virtual machines are dedicated for small footprint embedded systems (SFES) or more specifically, for 8- or 16 bit microcontrollers with a maximum memory size of 64K bytes. The virtual machine:

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

For 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. Also, our memory manager is central to all dynamic (object) allocations.

3 Abstraction: From C to OO

Historically, modeling and programming small embedded systems has been almost impossible to tackle with an object-oriented approach. Using classes and objects was simply too demanding for the limited space and processing power of the 8-bit microcontroller, the fundamental building block of embedded systems. For that reason (and a good reason at that), the majority of embedded systems programmers continue to use C as their programming language of choice. It is compact, it is quick, and it is widely-supported. However, by adopting C, we have no option but to adopt a traditional procedural approach to programming. Although C is also versatile and allows us to emulate more modern programming paradigms or styles, such as the object-based or object-oriented approach, doing so often requires sophisticated programming techniques. Yet, the motivation for developing collections in an object-oriented (OO) way supports one fundamental advantage: write once for all and reuse often.

In this paper, we examine the evolution of the three predominant paradigms of software development from the procedural to the object-based to the object-oriented approach. Although each has its advantages and disadvantages, the object-oriented paradigm is best able to support the concepts of information hiding and extensibility. These concepts have a major impact on the maintainability, reliability, and portability of embedded systems applications.

4 Procedural Paradigm

The first and traditional paradigm of programming is based on the principle of procedural abstraction. This paradigm divides developmental work into two distinct parts. First, real-world entities are identified and mapped as structures or records (data) and second, functions are written to act upon this data (behavior). Each function can be thought of as the proverbial black box, receiving data as input, processing it, and returning data as output. When embedded systems programmers were first able to write applications in a high-level programming such as C, they were introduced to the procedural approach. Applications were therefore decomposed into functions which were defined in terms of the data they received, manipulated, and returned.

The procedural approach suffers from two primary drawbacks. First, data and behavior are modeled separately. Hence, because data may be shared among several functions via global variables or parameters, responsibility for its behavior is scattered and open-ended. Data when divorced from its behavior supports neither the seminal principle of information hiding nor the notion of encapsulation. For this reason, applications using the procedural approach can be difficult to test, debug, and maintain which leads to its second

primary drawback: extensibility. Extensibility allows one to easily incorporate or build upon existing code without undo burden on the current application. Again, because data and behavior are modeled separately, the tentacles of code which manipulate certain data need to be tracked down when changes to data are made or when new data is introduced.

To illustrate these drawbacks, we consider the very small problem of writing an embedded application to read a value from a device `d`. Using the procedural approach, the function `Read` is designed to receive a certain device, fetch the proper value, and return it to the calling program. Its invocation may appear as follows.

```
value = Read(d);
```

Problems arise with the procedural approach when an application must access different kinds of devices, such as buttons, keypads, and sensors. In traditional languages like C, it is not possible to reuse the same function name for each kind of device; distinct function names are therefore required. This can be done by prefixing or suffixing the action we want (in our case, a `Read`) for each device: `ButtonRead`, `KeypadRead`, and `SensorRead`.

Once we have differentiated and compiled the various `Read` actions, we need to select the proper `Read` for a given device. The solution in traditional procedural languages often uses an enumeration of constants in order to make the correct selection. This enumeration must be publicly available, ideally in a header file, to be reused by the caller. In the following example, three kinds of devices (constants) are associated with `DeviceType`.

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

We also need a data structure to represent a device with one data member set aside to specify its type. This data member is initialized when a device is created.

```
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.
```

Now, in using any device `d`, careful tests on device type must be performed in order to invoke the proper corresponding read. The following is an example using an `if-then-else` statement:

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

or its equivalent using a multiple-choice `switch` statement:

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

Moreover, the above code is now statically bound to constants in `DeviceType`, meaning that any addition of a new kind of device or any removal of an existing device will likely require a massive recompilation (and static relinking) of all source files that use (or include in C) these device declarations. It quickly becomes a maintenance nightmare. All statements which test for device type may be scattered among thousands of lines of code and must be updated for each device that is added or removed. Therefore, the update is very prone to error. Here is an example of the impact of adding a `KEYBOARD` device to the above switch statement:

```
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 every `switch` statement must be modified for each reference to `d.type`.

The procedural approach was no doubt a major and requisite step forward in the evolution of program design. But it was only a partial solution. From the point of view of maintaining and reusing code efficiently, it is important to encapsulate data and the functions that manipulate it into a single, cohesive syntactic structure. It is important to hide away the implementation details of data and to strictly define data in terms of a consistent behavior and interface.

5 Object-Based Paradigm

The object-based paradigm is built on the principle of data abstraction. Unlike the procedural approach, this paradigm divides developmental work into two quite different tasks. First, the data *and* behavior of each real-world entity of the problem domain are identified and encapsulated into a single structure called a class. Second, objects drawn from the different classes are created and work together to provide a solution to the given problem. Fundamentally, each object is responsible for the behavior of its own data.

Based on our experience as teachers, coaches, and trainers in object-oriented technologies, the shift from a procedural to an object-based approach often passes through four phases before a certain level of programming comfort is reached. We identify these four phases as using concrete data structures (CDS), abstract data structures (ADS), abstract data types (ADT), and classes, respectively. Each phase has its own advantages and disadvantages and each phase may well be suited to the constraints of a particular embedded system. But on balance, the progression through the four phases moves from poor to fair to good to excellent use of data abstraction.

To illustrate this evolution from the procedural to object-based approach, we take the example of the ubiquitous stack. The `Stack` is a data type with at least four standard functions: `Stack_Init`, `Stack_Pop`, `Stack_Push`, and `Stack_IsEmpty`. These functions manipulate the stack in a last-in, first-out manner. Using a procedural approach, the data of the `Stack` is modeled first, in this case, as a struct with a two fields: an array `items` which holds up to `SIZE` integer values and an integer `sp` which stores the index of the top element.

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

Next, the functions which manipulate the `Stack` are modeled separately. For example, the function `Stack_Push`, is implemented below with appropriate parameters, including a formal parameter for the `Stack` to be modified.

```
void Stack_Push(Stack s, int item) {
    items[++sp] = item;
}
```

Under the procedural approach, however, there is no impediment for any programmer to add functions which take any parameter of type `Stack` and manipulate it in ways which violate the last-in, first-out principle. As mentioned in the previous section, this is one of the greatest drawbacks of the procedural approach.

We now take an evolutionary look at the four phases which moved programming practice from the procedural to the object-based paradigm, pointing out the benefits and pitfalls of each step.

6 Concrete Data Structures in C

The concept of a concrete data structure (CDS) is the first and most rudimentary attempt at data abstraction. Using this approach, all user-defined data structures are visible to all users. Although global access is very efficient, the data itself is vulnerable to unspecified changes from any user, violating the fundamental principle of information hiding and usurping consistent behavior. Also, code is married to a particular instance of data and therefore, code reuse is not possible.

6.1 Declaring a Concrete Data Structure

When a stack is implemented as a concrete data structure (CDS), its interface or header file exposes all declarations as global: the constant `SIZE`, the type `Stack`, the variable `s`, and the functions signatures for `Stack_Init`, `Stack_Pop`, `Stack_Push`, and `Stack_IsEmpty`.

```
/* StackCDS.h - CDS for a stack of integers */

const int SIZE = 10;

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

Stack s;

void  Stack_Init()         { ... }
int   Stack_Pop()          { ... }
void  Stack_Push(int item) { ... }
bool  Stack_IsEmpty()      { ... }
```

Notice that only one copy of stack is available in the application. It is not possible to create more than one.

6.2 Using a Concrete Data Structure

From a user's perspective, there is direct access to `Stack s`. Hence, changes to `s` can be made without using one of the four functions.

```
#include "StackCDS.h"

Stack_Init();
Stack_Push(10);

if ( !Stack_IsEmpty() ) {
    int n = Stack_Pop();  /* Good programming: via the interface     */
    n = s.items[ s.sp ];  /* Poor programming: direct access to data */
    ...
}
```

With this approach, users need be wary of possible modifications to the data, especially those that directly access `s` from elsewhere. Without any guarantee of consistent behavior, concrete data structures suffer the same drawback as the procedural approach: a lack of information hiding. In the next section on abstract data structures, the implementation details of the stack are hidden away from the user.

7 Abstract Data Structures in C

Like a concrete data structure (CDS), an abstract data structure (ADS) is a unit that integrates both data and functions. But unlike a CDS, the data is hidden within the unit and can only be accessed by its public functions. The data structure is called abstract because only the qualified names of the public functions are

known to the user (e.g. `Stack_Pop()`). The implementation details of the functions and the data itself are hidden. Hence, abstract data structures support information hiding, meaning that if the implementation changes, the interface remains unchanged. It also means that ADSs have a state that can only be modified by public functions. Therefore, the values of the internal data structure do not change between function invocations.

7.1 Declaring an Abstract Data Structure

In C, an abstract data structure (ADS) can be implemented by hiding internal constants and structures in a separate implementation file. We call this file `StackADS.c` in C. Only the stack functions are visible as part of the header file:

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

void  Stack_Init();
int   Stack_Pop();
void  Stack_Push(int item);
bool  Stack_IsEmpty();
```

7.2 Using an Abstract Data Structure

From a user's perspective, an abstract data structure is similar to a concrete data structure except that any attempt to directly access the internal data structure of `Stack` is flagged by the compiler as follows:

```
#include "StackADS.h"

Stack_Init();
Stack_Push(10);

if ( !Stack_IsEmpty() ) {
    int n = Stack_Pop();  /* Good programming: via the interface   */
    n = s.items[ s.sp ];  /* Compiler error: direct access to data */
}
```

Abstract data structures (ADSs) have several advantages:

- Users need not be familiar with or have access to implementation details.
- Implementation can be modified at any time without affecting the users.
- Data is protected in the implementation file (i.e. there is no global access).

ADSs also have some disadvantages:

- Accessing the data through functions is less efficient.
- The implementation file must be modified if a new operation (functionality) is needed to access the internal data.
- Only one instance of the data can be created.

In the next section on abstract data types, it is shown how multiple instances of data can be created.

8 Abstract Data Types in C

As its name implies, an abstract data type (ADT) is a unit that defines a type rather than a specific variable. Hence, more than one variable (or instance) of an abstract data type may be instantiated.

8.1 Declaring an Abstract Data Type

An abstract data type can be implemented in one of two ways: as a structure or as a pointer to a structure. In this section, we choose to implement the stack using a pointer to a structure. This pointer is an opaque type based on a forward reference to the data structure. The user therefore never sees or has access to the data structure directly, only indirectly via the pointer itself. The C interface for the abstract data type `Stack` is illustrated as follows:

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

        struct Stack_Desc;            /* Forward reference */
typedef struct Stack_Desc* Stack  ;   /* Opaque type */

Stack   Stack_New    ();
void    Stack_Init   (Stack this);
void    Stack_Push   (Stack this, int item);
int     Stack_Pop    (Stack this);
bool    Stack_IsEmpty(Stack this);
void    Stack_Delete (Stack this);
```

The corresponding implementation exports only the structure identifier (tag) called `Stack_Desc`. Its individual fields are hidden (not exported). Two additional functions, `Stack_New` and `Stack_Delete`, are defined to dynamically allocate and de-allocate each instance of `Stack`. Because abstract data types permit the creation of multiple instances, each variable of type `Stack` also has its own set of data. Finally, the C implementation forces all access functions to have one additional parameter called `this` that points to the particular stack to be modified.

8.2 Using an Abstract Data Type

Based on the interface of an abstract data type, users instantiate multiple variables of type `Stack` as follows:

```
#include "StackADT.h"

Stack aStack, anotherStack;

aStack       = Stack_New();    /* Stack_New() calls Stack_Init() internally */
anotherStack = Stack_New();

Stack_Push(aStack, 10);

if ( !Stack_IsEmpty(aStack) ) {
    int n = Stack_Pop(aStack);
    ...
}
```

Both `aStack` and `anotherStack` have their own copies of data, but are manipulated via their public functions in the same way. Hence, information hiding and consistent behavior are both ensured. Abstract data types (ADTs) therefore have two key advantages:

- New data types can be added to the language.
- New data types can be used like any other basic type, such as `char`.

ADTs also have some disadvantages:

- Accessing data is less efficient than abstract data structures because of the additional parameter.
- Dynamic allocations are less efficient than static allocations.

To summarize, the designer of an abstract data type must consider if multiple instances of an abstract data type are really required. Not all objects are best implemented as ADTs. Although some objects such as stacks, queues, and devices are naturally implemented as ADTs, other objects such as an LCD, keypad, or keyboard normally have only one instance and are more efficiently implemented as abstract data structures (ADSs).

8.3 Implementing an Abstract Data Type

The full implementation of `Stack` as an abstract data type is shown as follows.

```
/* StackADT.c - implementation of a stack of integers */

#include "StackADT.h"

const int SIZE = 10;

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

Stack Stack_New () {
    Stack s = (Stack) malloc(sizeof(Stack_Desc));
    Stack_Init( s );
    return s;
}

void Stack_Init(Stack s) {
    s->sp = -1;
}

void Stack_Push(Stack s, int item) {
    s->items[++(s->sp)] = item;
}

int  Stack_Pop (Stack s) {
    return s->items[(s->sp)--];
}

int  Stack_IsEmpty (Stack s) {
    return s->sp == -1;
}

void Stack_Delete(Stack s) {
    free(s);
}
```

9 Code Reuse in C with Wrappers

The reusability of code is enhanced if data is decoupled from its representation, that is, the representation of the data is opaque. The class structure in languages such as C++ is a wrapper which allows the developer to clearly separate the declaration and implementation of data structures. Opaque types allow changes in an implementation without affecting the class interface on which user code depends. This section presents a C++ class that completely encapsulates the implementation details of a generic C code package. It demonstrates also that true reusability is achieved by linking existing C packages in binary form.

In our case here, a typical class is presented that completely encapsulates the implementation details of of our previous collection C stack and hence ensures that the representation of data is opaque. Therefore, implementation structure details are hidden from the class declaration. A true reusability is achieved by

linking generic C packages in binary form, so designers need not rewrite existing reusable C code components in C++.

The designer is now free to change the data representation in a manner completely transparent to the user code. Since the implementation file encapsulates the data representation rather than the specification file, the user code is only relinked and not recompiled.

9.1 Using a C++ Wrapper

We now define a class `Stack` and its member functions in terms of existing functions. The data representation of the `Stack` is opaque and the private data in the class declaration is a pointer to an object of that opaque type. Hence, we have the ability to change the representation of the header without forcing the recompilation of users of that class.

```
// Stack.hpp - C++ Interface of a C Stack

#ifndef STACK_HPP
#define STACK_HPP

extern "C" {
    #include "StackADT.h" // C interface containing the opaque type "Stack_Desc"
}

class Stack {
public:
    Stack();
    void Push(int item);
    int  Pop ();
    int  IsEmpty ();

    ~Stack();

private:
    Stack_Desc* header; // Pointer to the opaque type "Stack_Desc"
};


Stack::Stack() {
    header = Stack_New();
}
Stack::~Stack() {
    Stack_Delete(header);
}
inline bool Stack::IsEmpty() {
    return Stack_IsEmpty(header);
}
#endif // STACK_HPP
```

Since most C functions require an argument list pointer, all methods can be written efficiently as inline functions in the interface file. The only exceptions are the constructor and destructor.

10 Classes in C++

Although an abstract data type can be implemented in a procedural language like C, it falls one step short of encapsulating data and behavior into a single syntactic unit. Instead, data and behavior are physically separated across two files. Using the object-oriented paradigm supported by the C++ language, data and functions are encapsulated into a single entity called a class.

The concepts of class and object are the backbone of the object-based, as well as the object-oriented, paradigm. Like an abstract data type, a class defines the blueprint of a new type, encapsulating both

representation (data) and behavior (functions) in a single syntactic unit. An object, on the other hand, is a particular instance of a class, much as a variable is an particular instance of a type. A class therefore supplies the mechanism to construct objects from its definition. Objects are manipulated by accessing the public functions defined in its corresponding class. Hence, the object-based approach can be distilled to a question of responsibility. In our example, the class `Stack` is responsible for its own behavior using its own functions to manipulate its own data. In short, a class is a generalization of an abstract data type, but is far easier to implement and maintain.

10.1 Declaring a Class

Let us see how the same stack, implemented as an abstract data type, can be implemented as a class in C++ as shown below. Notice that all data members (`items`, `sp`, and the constant `SIZE`) are private and hidden. This supports the notion of information hiding. Therefore, the only way to modify an instance of `Stack`, that is an object of type `Stack`, is to invoke one of its public functions (`Push`, `Pop`, `IsEmpty`, and so on), otherwise known as methods using object-based and object-oriented terminology. Finally, to simplify the allocation and deallocation of objects, each class in C++ generally includes two special methods called a constructor and destructor, respectively. The names of the constructor and destructor are the same as that of the class (i.e. `Stack`) except that the destructor is preceded by the tilde sign `~`.

```
/* Stack.bsharp - implementation of a stack of integers in C++ */

class Stack {
    public Stack() { // Constructor
        items = new int[ SIZE ];
        Init();
    }

    public void Init() {
        sp = -1;
    }

    public void Push(int item) {
        items[++sp] = item;
    }

    public int Pop() {
        return items[sp--];
    }

    public bool IsEmpty() {
        return sp == -1;
    }

    public ~Stack() { // Destructor
        delete items;
    }
    private const int SIZE = 10;
    private       int sp;
    private       int items[ SIZE ];
}
```

In C++, the order of declaration for both data members and methods is unimportant since the compiler makes multiple passes. In the C/C++ programming languages (and compilers), everything must be declared and exposed via header files before usage.

10.2 Using a Class

Syntactically, the `Pop` function (method) in C++ is invoked in the following way, assuming an instance of `Stack` called `s` has already been created:

```
s.Pop( );
```

Other methods of `Stack` are invoked in a similar way. Using the procedural paradigm, the C invocation of `Pop` as described earlier is done as follows:

```
Pop(s);
```

It is worthwhile to pause at this moment and reflect on the how the difference between procedural and data abstraction is manifest in the syntax of the language. Using the procedural paradigm, a function such as `Pop` accepts the data as a parameter and modifies it accordingly. Indeed, any function can accept the stack `s` as a parameter and thereby, modify its behavior arbitrarily. Under the object-based paradigm, modifications to the stack must use one of the methods that "belong" to the class `Stack`.

11 Conclusion

Over the last decades or so, the development of software applications has relied more and more heavily on code reuse. However, code reuse depends on two fundamental requisites: a uniform memory model that hides away the details of the underlying architecture and a class structure which encapsulates data and its behavior into a single syntactic unit. As we move from a procedural to a object-oriented 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. In this paper, we have demonstrated that C can be used successfully to implement abstract data types (ADTs), an important step forward.

Code reuse eliminates the re-implementation of common data structures such as stacks and queues. Although code reuse is possible in C, it is best exploited using an object-based approach and a uniform memory model.

References

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

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