Networked Embedded System Patterns for C Developers

Part II: Overview of C (& C++) Programming Styles

Douglas C. Schmidt

Professor
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/∼schmidt/

Department of EECS
Vanderbilt University
(615) 343-8197

June 7, 2009

Motivation

- C supports a range of programming styles
  - *e.g.*, data hiding, data abstraction, & object-oriented programming
- Different C programming styles yield different pros & cons for developers of embedded software
- The following slides examine various programming styles using a Stack to illustrating the pros & cons of evolving from simple to more abstract styles
- These different programming styles are based on different *patterns*
  - Patterns can be used to emulate OO design/programming in C

A Bare-bones Stack Implementation in C

- First, consider a "bare-bones" C implementation of a stack:

```
typedef int T;
/* enum { MAX_STACK = 100 }; */
#define MAX_STACK 100
T stack[MAX_STACK];
int top = 0;
T item = 10;
// push
stack[top++] = item;
...
// pop
item = stack[--top];
```

Evaluating the Bare-bones Implementation in C

- Pros

  1. No unnecessary run-time overhead ;-)

- Cons

  1. Solution is very tightly coupled to a particular implementation
  2. There is only *one* stack & only *one* type of stack
  3. Name space pollution
  4. Error handling is non-existent
  5. Solution is non-reentrant

Data Hiding Implementation in C

- Define the interface to a Stack of integers in C in Stack.h:

```
/* Type of Stack element. */
typedef int T;

/* Stack interface. */
int create (size_t size);
int destroy (void);
void push (T new_item);
void pop (T *old_top);
void top (T *cur_top);
int is_empty (void);
int is_full (void);
```

Data Hiding Implementation in C (cont'd)

- /* File stack.c */

```
#include "stack.h"
static int top_, size_; /* Hidden within this file. */
static T *stack_;
int create (size_t size) {
  top_ = 0; size_ = size;
  stack_ = malloc (size * sizeof (T));
  return stack_ == 0 ? -1 : 0;
}
void destroy (void) { free ((void *) stack_); }
void push (T item) { stack_[top_++] = item;}
void pop (T *item) { *item = stack_[--top_]; }
void top (T *item) { *item = stack_[top_ - 1]; }
int is_empty (void) { return top_ == 0; }
int is_full (void) { return top_ == size_; }
```

Applying the Data Hiding Implementation in C

```
#include "stack.h"
void foo (void) {
  T i;
  create (100);
  if (is_full () == 0)
    push (10);
  if (is_full () == 0)
    push (20);
  if (is_empty () == 0)
    pop (&i);
  destroy ();
}
```

Evaluating the Data Hiding Implementation in C

- Pros

  1. Implementation details are hidden from client applications

- Cons

  1. The programmer must call `create()` first & `destroy()` last!
  2. There is only *one* stack & only *one* type of stack
  3. Name space pollution
  4. Error handling is overt/awkward
  5. Non-reentrant

Data Abstraction Implementation in C

- An ADT Stack interface in C:

```
typedef int T;
typedef struct { size_t top_, size_; T *stack_; } Stack;

int Stack_create (Stack *s, size_t size);
void Stack_destroy (Stack *s);
void Stack_push (Stack *s, T item);
void Stack_pop (Stack *, T *item);
/* Must call before pop'ing */
int Stack_is_empty (Stack *);
/* Must call before push'ing */
int Stack_is_full (Stack *);
/* ... */
```

Data Abstraction Implementation in C (cont'd)

- An ADT Stack implementation in C:

```
#include "stack.h"
int Stack_create (Stack *s, size_t size) {
  s->top_ = 0; s->size_ = size;
  s->stack_ = (T *) malloc (size * sizeof (T));
  return s->stack_ == 0 ? -1 : 0;
}
void Stack_destroy (Stack *s) {
  free ((void *) s->stack_);
  s->top_ = 0; s->size_ = 0; s->stack_ = 0;
}
void Stack_push (Stack *s, T item)
{ s->stack_[s->top_++] = item; }
void Stack_pop (Stack *s, T *item)
{ *item = s->stack_[--s->top__]; }
int Stack_is_empty (Stack *s) { return s->top_ == 0; }
```

Applying the Data Abstraction Implementation in C

```
void foo (void) {
  Stack s1, s2, s3; /* Multiple stacks! */
  T item;

  Stack_pop (&s2, &item); /* Pop'd empty stack */

  /* Forgot to call Stack_create! */
  Stack_push (&s3, 10);

  s2 = s3; /* Disaster due to aliasing!!! */

  /* Destroy uninitialized stacks! */
  Stack_destroy (&s1); Stack_destroy (&s2);
}
```

Evaluating the Data Abstraction Implementation in C

- Pros

  1. Can have multiple stacks
  2. Better control over the name space

- Cons

  1. No guaranteed initialization, termination, or assignment
  2. Still only one type of stack supported
  3. Too much overhead due to function calls
  4. No generalized error handling...
  5. The C compiler does not enforce information hiding *e.g.*,

```
s1.top_ = s2.stack_[0]; /* Violate abstraction */
s2.size_ = s3.top_; /* Violate abstraction */
```

Another C Data Abstraction Implementation

- An ADT Stack interface in C:

```
typedef int T;
typedef struct Stack *Stack;

int Stack_create (Stack *s, size_t size);
void Stack_destroy (Stack s);
void Stack_push (Stack s, T item);
void Stack_pop (Stack s, T *item);
/* Must call before pop'ing */
int Stack_is_empty (Stack s);
/* Must call before push'ing */
int Stack_is_full (Stack s);
/* ... */
```

Another C Data Abstraction Implementation (cont'd)

```
#include "stack.h"
struct Stack { size_t top_; size_t size_; T *stack_; };

int Stack_create (Stack *s, size_t size) {
  Stack_Impl *si = (Stack_Impl *) malloc (sizeof (Stack_Impl));
  if (si == 0) return -1;
  si->top_ = 0; si->size_ = size; si->stack_ = (T *) malloc (size * sizeof (T));
  if (s->stack_ == 0) return -1;
  *s = (Stack) si; return 0;
}
void Stack_destroy (Stack s) {
  Stack_Impl *si = (Stack_Impl *) s;
  free ((void *) ss->stack_);
  si->top_ = 0; si->size_ = 0; si->stack_ = 0;
  free ((void *) si);
}
```

Another C Data Abstraction Implementation (cont'd)

```
void Stack_push (Stack s, T item) {
  Stack_Impl *si = (Stack_Impl *) s;
  si->stack_[si->top_++] = item;
}

void Stack_pop (Stack s, T *item) {
  Stack_Impl *si = (Stack_Impl *) s;
  *item = si->stack_[--si->top__];
}

int Stack_is_empty (Stack s) {
  Stack_Impl *si = (Stack_Impl *) s;
  return si->top_ == 0;
}
```

Applying Another C Data Abstraction Implementation

```
void foo (void) {
  Stack s1, s2, s3; /* Multiple stacks! */
  T item;

  Stack_create (&s1, 100);
  Stack_pop (s2, &item); /* Pop'd empty stack */

  /* Forgot to call Stack_create! */
  Stack_push (s3, 10);

  s2 = s3; /* Disaster due to aliasing!!! */

  Stack_destroy (s1);
  Stack_destroy (s2);  Stack_destroy (s3);   /* Destroy uninitialized stacks! */
}
```

Evaluating the Other C Data Abstraction Implementation

- Pros

  1. Better data abstraction than the previous version

- Cons

  1. No guaranteed initialization, termination, or assignment
  2. Still only one type of stack supported
  3. More overhead due to function calls & dynamic memory allocation
  4. No generalized error handling...

Data Abstraction Implementation in C++

- We can get encapsulation, more than one stack, more than one type, guaranteed initialization/termination/assignment, & inlining:

```
template <typename T> class Stack {
public:
  Stack (size_t size);
  Stack (const Stack &s);
  void operator= (const Stack &);
  ~Stack (void);
  void push (const T &item);
  void pop (T &item);
  bool is_empty (void) const;
  bool is_full (void) const;
private:
  size_t top_, size_;
  T *stack_;
};
```

Data Abstraction Implementation in C++ (cont'd)

```
template <typename T>
Stack<T>::Stack (size_t s): top_ (0), size_ (s), stack_ (new T[s]) {}

template <typename T>
Stack<T>::Stack (const Stack<T> &s)
  : top_ (s.top_), size_ (s.size_), stack_ (new T[s.size_]) {
  for (size_t i = 0; i < s.size_; i++) stack_[i] = s.stack_[i];
}

template <typename T>
void Stack<T>::operator = (const Stack<T> &s) {
  if (this == &s) return; T *temp_stack = new T[s.size_]; delete [] stack_;
  for (size_t i = 0; i < s.size_; i++) temp_stack[i] = s.stack_[i];
  stack_ = temp_stack; top_ = s.top_; size_ = s.size_;
}
```

Data Abstraction Implementation in C++ (cont'd)

```
template <typename T>
Stack<T>::~Stack (void) { delete [] stack_; }

template <typename T>
bool Stack<T>::is_empty (void) const { return top_ == 0; }

template <typename T>
bool Stack<T>::is_full (void) const { return top_ == size_; }

template <typename T>
void Stack<T>::push (const T &item) { stack_[top_++] = item; }

template <typename T>
void Stack<T>::pop (T &item) { item = stack_[--top_]; }
```

Applying the C++ Data Abstraction Implementation

```
#include "Stack.h"
void foo (void) {
  Stack<int> s1 (1), s2 (100);
  int item;
  if (!s1.is_full ())
    s1.push (473);
  if (!s2.is_full ())
    s2.push (2112);
  if (!s2.is_empty ())
    s2.pop (item);
  // Access violation caught at compile-time!
  s2.top_ = 10;
  // Termination is handled automatically.
}
```

Evaluating the C++ Data Abstraction Implemenation

- Pros
  1. Data hiding & data abstraction, *e.g.*,
     ```
     Stack<int> s1 (200); s1.top_ = 10 // Error flagged by compiler!
     ```
  2. Can declare multiple stack objects of multiple types
     ```
     Stack<int> s1 (10); Stack<char> s2 (20); Stack<double> s3 (30);
     ```
  3. Automatic initialization & termination
     ```
     {
       Stack<int> s1 (1000); // constructor called automatically.
       // ...
       // Destructor called automatically
     }
     ```
- Con - doesn't have generalized error handling mechanism

Optimizing Template Implementations in C++

- Another parameterized type Stack class

```
template <typename T, size_t SIZE> class Stack {
public:
  Stack (void);
  void push (const T &item);
  void pop (T &item);
private:
  size_t top_, size_;
  T stack_[SIZE];
};
```

- Note, there's no longer any need for dynamic memory, though SIZE must be a constant, *e.g.*,

```
Stack<int, 200> s1;
```

Object-Oriented (OO) Implementation in C++

- Problems with previous examples:
  - Changes to the implementation will require recompilation & relinking of clients
  - Extensions will require access to the source code
- Solutions
  - Combine inheritance with dynamic binding to *completely* decouple interface from implementation & binding time
  - This requires the use of C++ *abstract base classes*

OO Implementation in C++ (cont'd)

- Defining an abstract base class in C++

```
template <typename T>
class Stack {
public:
  virtual ~Stack (void) = 0; // Need implementation!
  virtual void push (const T &item) = 0;
  virtual void pop (T &item) = 0;
  virtual bool is_empty (void) const = 0;
  virtual bool is_full (void) const = 0;
  void top (T &item) { // Template Method
    pop (item); push (item);
  }
};
```

- By using "pure virtual methods," we can guarantee that the compiler won't allow instantiation!

OO Implementation in C++ (cont'd)

- Inherit to create a specialized stack implemented via an STL vector:

```
#include "Stack.h"
#include <vector>

template <typename T> class V_Stack : public Stack<T> {
public:
  enum { DEFAULT_SIZE = 100 };
  V_Stack (size_t size = DEFAULT_SIZE);
  virtual void push (const T &item);
  virtual void pop (T &item);
  virtual bool is_empty (void) const;
  virtual bool is_full (void) const;
private:
  size_t top_; // built-in
  std::vector<T> stack_; // user-defined
};
```

OO Implementation in C++ (cont'd)

- class `V_Stack` implementation

```
template <typename T>
V_Stack<T>::V_Stack (size_t size): top_ (0), stack_ (size) {}

template <typename T> void
V_Stack<T>::push (const T &item) { stack_[top_++] = item; }

template <typename T> void
V_Stack<T>::pop (T &item) { item = stack_[--top_]; }

template <typename T> int
V_Stack<T>::is_full (void) const
{ return top_ >= stack_.size (); }
```

OO Implementation in C++ (cont'd)

- Inheritance can also create an linked list stack:

```
template <typename T> class Node; // forward declaration.
template <typename T> class L_Stack : public Stack<T> {
public:
  enum { DEFAULT_SIZE = 100 };
  L_Stack (size_t hint = DEFAULT_SIZE);
  ~L_Stack (void);
  virtual void push (const T &new_item);
  virtual void pop (T &top_item);
  virtual bool is_empty (void) const { return head_ == 0; }
  virtual bool is_full (void) const { return 0; }
private:
  // Head of linked list of Node<T>'s.
  Node<T> *head_;
};
```

OO Implementation in C++ (cont'd)

- class Node implementation

```
template <typename T> class Node {
friend template <typename T> class L_Stack;
public:
  Node (T i, Node<T> *n = 0): item_ (i), next_ (n) {}
private:
  T item_;
  Node<T> *next_;
};
```

- Note that the use of the "Cheshire cat" idiom allows the library writer to completely hide the representation of class `L_Stack`...

OO Implementation in C++ (cont'd)

- class L_Stack implementation:

```
template <typename T> L_Stack<T>::L_Stack (size_t): head_ (0) {}

template <typename T> void L_Stack<T>::push (const T &item) {
  Node<T> *t = new Node<T> (item, head_); head_ = t;
}

template <typename T> void L_Stack<T>::pop (T &top_item) {
  top_item = head_->item_;
  Node<T> *t = head_; head_ = head_->next_;
  delete t;
}

template <typename T> L_Stack<T>::~L_Stack (void)
{ for (T t; head_ != 0; pop (t)) continue; }
```

Evaluating C++ OO Implementation

- Using our abstract base class, it is possible to write code that does not depend on the stack implementation, *e.g.*,

```
template <typename T> Stack<T> *make_stack (bool use_V_Stack)
{ return use_V_Stack ? new V_Stack<T> : new L_Stack<T>; }

template <typename T>
void foo (Stack<T> *stack, const T &t) {
  T s;
  stack->push (t);
  stack->pop (s);
  // ...
}

foo<int> (make_stack<int> (true), 10);
```

Evaluating C++ OO Implementation (cont'd)

- Moreover, we can make changes at run-time without modifying, recompiling, or relinking existing code
  - *i.e.*, can use "dynamic linking" to select stack representation at run-time, *e.g.*,

```
char stack_symbol[MAXNAMLEN];
char stack_file[MAXNAMLEN];
cin >> stack_file >> stack_symbol;
void *handle = ACE_OS::dlopen (stack_file);
void *sym = ACE_OS::dlsym (handle, stack_symbol);
if (Stack<int> *sp = // Note use of RTTI
  dynamic_cast <Stack<int> *> (sym)) foo (sp);
```

- Note, no need to stop, modify, & restart an executing application!
  - Naturally, this requires careful configuration management...

OO Implementation in C

- Some platforms (*e.g.*, COM & BREW) mimic OO programming in C

```
typedef int T;
AEEINTERFACE(IStack) {
  void (*Push)(IStack *po, T item);
  void (*Pop)(IStack *po, T *item);
  int  (*Is_Empty)(IStack *po);
  int  (*Is_Full)(IStack *po);
  void (*Delete)(IStack *po);
};

#define ISTACK_Push(p,item) AEEGETPVTBL(p,IStack)->Push(p, item)
#define ISTACK_Pop(p,item)  AEEGETPVTBL(p,IStack)->Pop(p, item)
#define ISTACK_Is_Empty(p)  AEEGETPVTBL(p,IStack)->Is_Empty(p)
#define ISTACK_Is_Full(p)   AEEGETPVTBL(p,IStack)->Is_Full(p)
#define ISTACK_Delete(p)    AEEGETPVTBL(p,IStack)->Delete(p)
```

OO Implementation in C (cont'd)

```
#define MP_ISTACK_SETVTBL(pVtbl, pfnPu, pfnPo, pfnIE, pfnIF, pfnDe) \
   (pVtbl)->Push     = (pfnPu); \
   (pVtbl)->Pop      = (pfnPo); \
   (pVtbl)->Is_Empty = (pfnIE); \
   (pVtbl)->Is_Full  = (pfnIF); \
   (pVtbl)->Delete   = (pfnDE); \

#define INHERIT_Stack(iname) \
   DECLARE_VTBL(iname)

struct Array_Stack {
  INHERIT_Stack(IStack);
  size_t top_
  size_t size_;
  T *stack_;
};
```

OO Implementation in C (cont'd)

```
static IStack *Array_Stack_Create (size_t size);
static void    Array_Stack_Push (IStack *po, T item);
static void    Array_Stack_Pop (IStack *po, T *item);
static int     Array_Stack_Is_Empty (IStack *po);
static int     Array_Stack_Is_Full (IStack *po);
static int     Array_Stack_Delete (IStack * po);

static IStack *Array_Stack_New (int16 nSize, VTBL(IStack) *pvt) {
  VTBL(IStack) *pVtbl;
  Array_Stack *pme = MALLOC (nSize + sizeof (VTBL(IStack)));
  if (!pme) return 0;
  pVtbl = (VTBL(IStack) *)((byte *) pme + nSize);
  MEMCPY (pVtbl, pvt, sizeof (VTBL(IStack)));
  INIT_VTBL (pme, IStack, *pVtbl);
  return (IStack *) pme;
}
```

OO Implementation in C (cont'd)

```
static IStack *Array_Stack_Create (size_t size) {
  Array_Stack *pme;
  VTBL(IStack) vtbl;

  MP_ISTACK_SETVTBL (&vtbl, Array_Stack_Push, Array_Stack_Pop, Array_Stack_Is_Empty,
                     Array_Stack_Is_Full, Array_Stack_Delete);
  pme = (Array_Stack *) Array_Stack_New (sizeof(Array_Stack), &vtbl);
  if (!pme) return NULL;
  else {
    pme->top_ = 0;
    pme->size_ = size;
    pme->stack_ = MALLOC (size * sizeof (T));
  }

  return (IStack *) pme;
}
```

OO Implementation in C (cont'd)

```
static void Array_Stack_Push(IStack *po, T item) {
  Array_Stack *pme = (Array_Stack *) po;
  pme->stack_[pme->top_++] = item;
}

static void Array_Stack_Pop(IStack *po, T *item) {
  Array_Stack *pme = (Array_Stack *) po;
  *item = pme->stack_[--pme->top_];
}
```

OO Implementation in C (cont'd)

```
static int Array_Stack_Is_Empty(IStack *po) {
  Array_Stack *pme = (Array_Stack *) po;
  return pme->top_ == 0;
}

static int Array_Stack_Is_Full(IStack *po) {
  Array_Stack *pme = (Array_Stack *) po;
  return pme->top_ >= pme->size_;
}

static void Array_Stack_Delete(IStack *po) {
  Array_Stack *pme = (Array_Stack *) po;
  FREE (pme->stack_);
  FREE (pme);
}
```

Applying OO Implementation in C

- Example application

```
IStack *is = Array_Stack_Create (100);
if (ISTACK_Is_Full (is) == 0)
  ISTACK_Push (is, 411);
int item;
if (ISTACK_Is_Empty (is) == 0)
  ISTACK_Pop (is, &item);
ISTACK_Delete (is);
```

Evaluating OO Implementation in C

- Pros

  1. Emulates OO programming using macros/patterns
  2. Relatively abstract programming model
  3. Can support release-to-release binary compatibility

- Cons

  1. No guaranteed initialization, termination, or assignment
  2. Still only one type of stack supported
  3. Too much overhead due to function calls

Concluding Remarks

- C (and C++) support a range of programming styles
  - *e.g.*, data hiding, data abstraction, object-oriented programming
- A major contribution of C++ is its support for defining abstract data types (ADTs) & for generic programming
  - *e.g.*, *classes*, *parameterized types*, & *exception handling*
- For some systems, C/C++'s ADT support is more important than using the OO features of the languages
- For other systems, the use of OO features is essential to build highly flexible & extensible software
  - *e.g.*, *inheritance*, *dynamic binding*, & *RTTI*
- Patterns can be used to emulate OO design/programming in C