By Edward F Binder, Motorola Computer Group

YOU CAN DESIGN SOFTWARE WITH OBJECT-ORIENTED TECHNIQUES USING A FAMILIAR LANGUAGE, SUCH AS ANSI-STANDARD C. YOU'LL BE ABLE TO EXERCISE DISCIPLINE IN CODE ORGANIZATION AND NAMING CONVENTIONS WITHOUT LEARNING NEW SYNTAX, LANGUAGE IDIOSYNCRASIES, OR DEBUGGING METHODS.

Implementing object-oriented designs in ANSI-standard C

In the 1990s, object-oriented analysis (OOA) and object-oriented design (OOD) offered software designers the promise of improved code expandability and reusability. These techniques also offered portability and reduced maintenance with early-life-cycle error discovery and correction. To promote implementation of these new design techniques, the software industry fostered the development of object-oriented-programming (OOP) languages. Some of the best-known of these languages are Smalltalk; Eiffel; Objective C and Pascal; later revisions of Ada; and the most widely used language, C++.

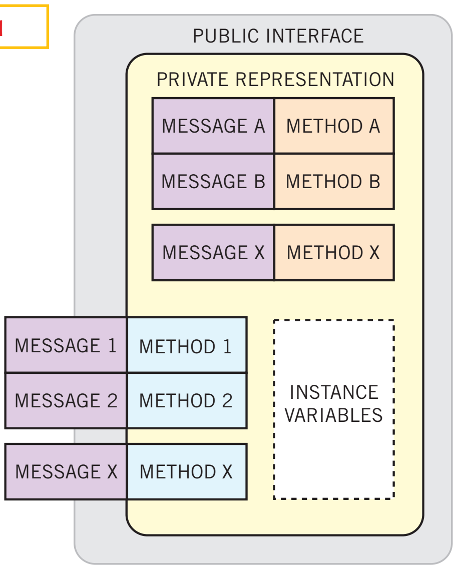
OOP languages provide a unique, sometimes proprietary, interpretation of typical OOD components, making implementation of object-based designs a basic process of translating class/object diagrams into a high-level OOP language. However, this benefit includes several drawbacks, which slowed industry acceptance of the new languages and the OOD methodology.

Critics often cited poor code performance due to the increased language overhead and resource demands needed for a complete polymorphic object model. The interpretive nature of these early languages was largely to blame for their so-called inefficiency. As hardware performance improved and the required storage resources became increasingly cost-effective, efficiency concerns evaporated, and C++ emerged as the software industry's first choice in OOP languages.

With its roots in ANSI C, C++ gained popularity by extending its predecessor's familiar form and style. C++ appealed to the broad base of C-programming talent, beating out syntactically foreign languages, such as Ada and Eiffel, which were often mired in proprietary-licensing complexities, large resource requirements, and extensive learning periods.

Although C++ is now in wide use, it still presents

Figure 1



A properly encapsulated object has a distinct public interface that hides the object's internal, private representation and structure.

new challenges to even the most seasoned ANSI-C software designer. Because the language's powerful object-oriented operation is hidden and dynamic, debugging methods are frequently obscure and time-consuming. Compiler variations introduce subtle operational and declarative idiosyncrasies, which you must overcome, sometimes with substantial creativity and in-depth understanding. Mastering the object-oriented extensions and understanding new language behaviors requires training, so if you have experience with only standard C and you lack the support of previously developed C++ class libraries, C++ may be unsuitable for your project. However, you may still use OOD as a methodology or as an implementation style.

OOD provides logical benefits regardless of the chosen programming language. You can use it with almost any syntax—even those syntaxes lacking object-oriented specifics. You cannot exploit all OOP features when using a typical procedural language, but you can achieve many organizational advantages. By following a few straightforward rules and conventions, you can achieve a logically cohesive, loosely coupled, object-based design that facilitates reuse, future feature expansion, and substantial error localization and isolation. Moreover, you can apply the methodology and coding technique to all levels of software—from end-user applications to device drivers.

When you couple implementation-independent techniques with design experience in a procedural language, you are on your way to well-organized object-based code. You need only visualize and construct the object components of your programming solution within the restrictions of a language whose syntax and behavior is already known and understood. Some challenge remains in accomplishing this task, but beneficial results come quickly without the extended learning periods necessary for mastering true object-oriented languages.

LISTING 1—CLASS-TYPE DECLARATION

```
    #ifndef _pipe_h_
#define _pipe_h_
…                                        Component
#include "buffer.h"                      Classes
#include "standard.h"
#include "status.h"
…                                        Class Attribute
#define PIPE_MAX_ID 25                   Declaration
…
…                                        Class Aggregation
typedef struct pipeClass                 Relationship
{
   Buffer_Pointer   itemBuffer;
   LongInteger      messageSize;
   LongInteger      pipeSize;
   LongInteger      queuedCount;
   char            *name;
   …                                     Allocated Object
} Pipe_Class;                            Reference Type
…
typedef Pipe_Class *Pipe_Pointer;
…
(continued, pipe.h)
```

Because of its near-universal acceptance and the depth of its user base, ANSI-standard C is an excellent choice for implementing object-based designs. The following guidelines show how to create the main components of the object-oriented methodology with standard C syntax and without adversely increasing execution overhead or code complexity. These guidelines may also help you to write coding standards and to later port object-based designs in C to C++.

THE COMPONENTS OF OOD

OOD emphasizes encapsulation of information and operations to form a single conceptual unit. Such a unit, or object, has a distinct public interface and private representation (**Figure 1**). This principle of information hiding separates the publicized ability to perform a specific task from the private steps necessary to accomplish the task. It allows abstraction of objects during design and for later internal improvement, extension, and error correction of the object without affecting the surrounding system.

Objects access each other by sending messages allowed by each object's public interface. When an object receives a message, the object performs the requested operation. The set of messages to which a given object can respond defines the object's behavior. However, all messages need not be a part of the object's public interface. An object can privately send messages to itself as necessary to carry out its public operations. A message comprises an operation name and any required parameters.

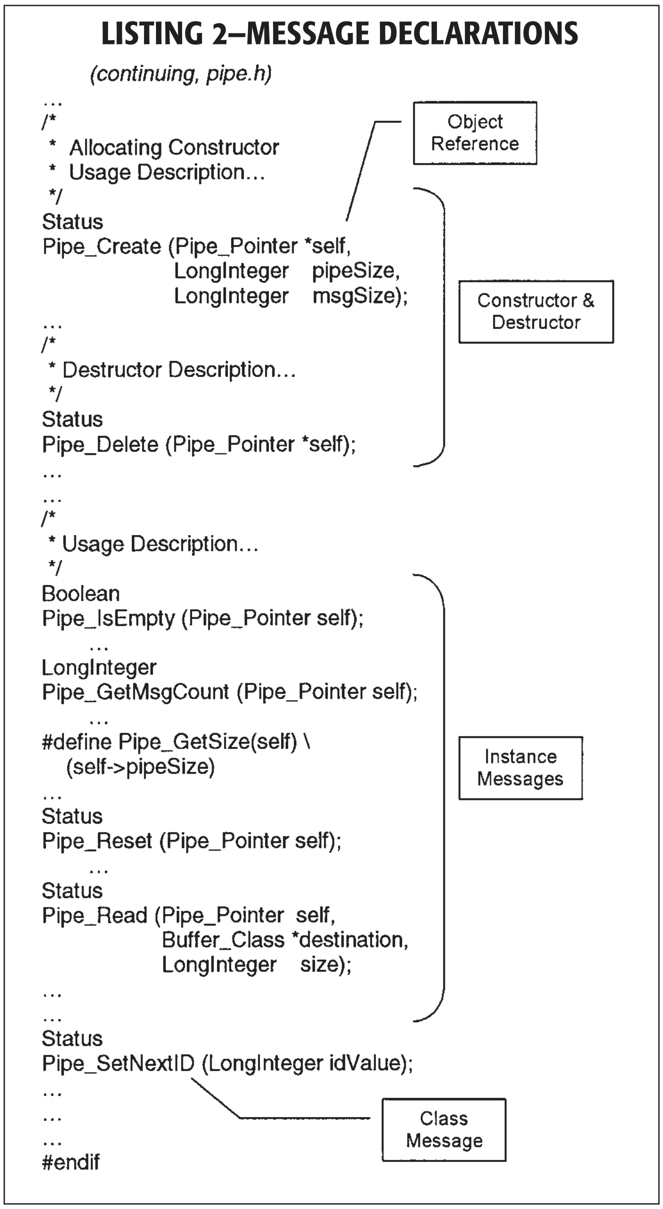
A method is the algorithm that the object executes in response to receiving a message. It is part of an object's private representation. The method name always matches the corresponding message name. When an operation has methods on multiple object types, all such methods should have a consistent intent and the same message signature across types. The signature is the formal message specification, comprising the type, the order and number of parameters, and the resultant value type. Consistent method intent and message signature over multiple object types allows you to group and classify objects into conceptual hierarchies of common properties and behavior.

Objects that share the same attributes and behaviors belong to the same class. A class is a generic specification, or template, for producing an object at an abstract, conceptual level. Objects derive their identities from variations in their class values and associations with other objects. Each object's private instance variables maintain the object's attribute values.

Classes that do not create instances of themselves are called abstract. They exist so that you can extract and collect common behavior among classes in one location. An abstract class specifies its default behavior so that subclass methods will refine, augment, or overload its implementations. A concrete subclass adds new abilities to the behaviors inherited from its abstract superclass as the subclass's function and purpose require. You must always fully implement a concrete class, because the class creates object instances of itself to perform within a software system.

Although inheritance allows you to create object classes by extending the behavior and structure of superclasses, you may also associate classes through aggregation to form new, more complex assembly classes. You associate classes representing the components of an object to form a class representing the entire assembly. This concept allows you to create complex objects by combining simpler, lesser objects and eases testing and error isolation. Simpler objects become easier to debug and, once these objects are operational, you can reuse them to build entire systems.

LISTING 2—MESSAGE DECLARATIONS



```
   (continuing, pipe.h)
...
/*
 *  Allocating Constructor
 *  Usage Description...
 */
Status
Pipe_Create (Pipe_Pointer *self,
             LongInteger   pipeSize,
             LongInteger   msgSize);
...
/*
 * Destructor Description...
 */
Status
Pipe_Delete (Pipe_Pointer *self);
...
...
/*
 * Usage Description...
 */
Boolean
Pipe_IsEmpty (Pipe_Pointer self);
     ...
LongInteger
Pipe_GetMsgCount (Pipe_Pointer self);
     ...
#define Pipe_GetSize(self) \
   (self->pipeSize)
...
Status
Pipe_Reset (Pipe_Pointer self);
     ...
Status
Pipe_Read (Pipe_Pointer  self,
           Buffer_Class *destination,
           LongInteger   size);
...
...
Status
Pipe_SetNextID (LongInteger idValue);
...
...
...
#endif
```

Object encapsulation and inherited class properties allow for polymorphism, a powerful object-oriented concept. It provides a mechanism for recognizing and exploiting similarities among object classes by allowing objects from two or more classes to respond to the same message. The sending object need not be characteristically aware of the object receiving the message. However, it must be aware that the receiver will execute a corresponding method appropriate for its class. So, objects of classes derived from the same superclass respond to the same inherited messages (because the signatures match) but do so in a manner that their class methods define.

Unfortunately, effective use of polymorphism requires substantial support from the implementation language, making an OOP language, such as C++, a necessity. You can code for polymorphism using a procedural language, but the limited results may not be worth the effort. Such implementations involve using case statements and variant records within superclass methods yielding increased program complexity and obscurity. These classes must be aware a priori of all subclasses that you will derive, thus limiting their general and long-term usefulness. Considering these drawbacks, polymorphism is best reserved for use with real OOP languages.

However, you should keep polymorphism in mind when designing classes and class hierarchies, even though you will ultimately simulate the system in a conventional procedural language. Polymorphic concepts can lead to more effective class decompositions and associations. Following such concepts can also simplify later porting of the object-based design to a true OOP language.

Once you select ANSI C as the implementation language, you must adopt some new disciplines and conventions to overcome C's global-namespace limitation and its lack of private data structures to implement the most basic object-based systems. You must also restrict or eliminate the use of global accesses. Object-based designs rarely, if ever, require global data or embedded external declarations. Proper encapsulation, naming conventions, class specifications, and respect for private attribute data make free-floating functions and global data unnecessary for most applications, even when you are using a weakly typed language, such as C.

You can effectively specify object classes by following a few general rules that you can directly apply to C constructs. Classes readily correspond to source-file pairs in C. The header file contains the public-message specifications and attribute-data declaration to form the class interface. The corresponding definition file contains the implementations of all public and private methods and maintains any internal class-data values. The file name should also be the class name; it is usually a singular noun. The name is plural only when it represents a collection. Because of C's global-function name space, you must use the class name as a prefix to all other publicly visible message and data declarations in the class to guarantee distinction.

ATTRIBUTE AND MESSAGE DECLARATIONS

The attribute-data declaration in the header file is the main class type. You should name the declaration using a class name with the suffix "class." The declaration is generally a compound data structure containing the instance variables that you use to define individual objects of the class. It can contain state information, other objects, relational information, and any information necessary to maintain the object. You should always treat this structure privately. Unlike C++, C does not enforce this privacy, so programming discipline is necessary to avoid creating unwanted data dependencies. If you plan to dynamically allocate the structure, you should define a reference type with the suffix "pointer" and use it as the primary public type. **Listing 1** shows a partial header file declaring a class of software queue called "pipe."

A class specification uses two general message types. First is the instance message, which always contains a reference to the specific object (instance) that receives the message. Next is the class message, which the class receives with no object references. When you implement these message types, you provide access to attribute data only through messages in the class's public interface. Messaging must be adequate to provide any required attribute manipulation or visibility and keep private the form of the data.

LISTING 3—CLASS AS A SINGULAR INSTANCE

```
    #ifndef _pipeTable_h_
#define _pipeTable_h_
...
#include "pipe.h"
#include "standard.h"
#include "status.h"
...
/*
 * Usage Description...
 */
Status
PipeTable_Initialize (LongInteger tableSize);
...
Status
PipeTable_Add (Pipe_Pointer tableItem);
...
Status
PipeTable_Retrieve (Pipe_Pointer *tableItem);
...
Boolean
PipeTable_IsFull (void);
...
LongInteger
PipeTable_GetAvailable (void);
...
...
#endif
```

Usage Description Precedes Each Message Declaration

No Instance Parameters Defined or Used with Any Message

Each instance message should access or modify at least one instance-attribute value. The message contains a parameter referencing the specific object. You should not access class attributes. Each class message should access or modify at least one class-attribute value. The message does not contain a parameter referencing an object. You should not access instance-data values. No message of a class should operate on objects of different classes unless the class is an aggregate or relational association of classes. In those cases, some messages require arguments of all related class types.

In true OOP languages, you implicitly use the object reference that instance messages require. In C++, such a reference is called the "this" pointer. Because C has no implicit passing or usage of parameters, you must declare the object reference as a formal parameter in each instance message. Borrowing from SmallTalk, you should name this parameter "self" in all classes, and the name should declare it to be of the class's primary public type (for example, "class" or "pointer").

Message names should be short verb phrases. Direct, concise names precipitate smaller and simpler method implementations. Accurate naming promotes "self-documenting" code, which requires less supporting commentary within the code flow. Operations with similar intent in different classes should use the same verb across those classes, allowing the prefixed class name to provide the necessary name distinction.

The following list contains some useful names for common operations and their functions:

- Reset—Use to return all class or object attributes to specific defaults.
- Set—Use as a prefix to an attribute name to make an attribute true or a given value.
- Get—Use as a prefix to an attribute name to return an attribute value without changing it.
- Is—Use as a prefix to a Boolean determination to return an attribute's condition or object state.
- Clear—Use as a prefix to an attribute name to zero or make false an attribute.
- Open, close—Use to begin or end the complete use of an object.
- Add, insert—Use to add an object at an available location within a collection or at a given location.
- Remove, replace—Use to delete an object from a collection or to add another object to the same location.
- Read, store, write, retrieve—Use to input or output persistent objects or collections.
- Make, break—Use to set or return all object or class attributes with one function call.

Because C, unlike C++, has no implicit function execution upon object creation, you must declare object constructors and destructors and then explicitly execute them when an object enters and exits its scope of use. You should name a constructor that dynamically allocates an object, returning an object reference, "create." You should name a constructor that readies a statically declared object before using "initialize." You should name the object destructor "delete." You must create or initialize an object before any other usage or reference, and you should delete the object when the system no longer needs it.

Message declarations are ordinary function prototypes and macros that you use like functions. Functions are generally better for debugging, but macros are more efficient for simple, instance-variable accesses. Macros keep instance data private from the external code without the execution overhead of a full function call. Messages generally return an error status unless the function returns a Boolean state or a simple number. The pipe-class specification continuing in **Listing 2** shows some illustrative message declarations.

If a class specification consists of class messages only, the class is a singular instance that internally maintains all defining attribute data. Usually, you statically declare the attribute structure, but the instance may have dynamically allocated component objects that require runtime initialization by an appropriate class message. A look-up table is an example of such an object. Its class messages store, retrieve, match items, and return table status while hiding the actual form of the table structure. Only one table is available in the system. **Listing 3** shows a partial specification for a table of pipes.

A class-specification header file typically contains the ancillary type declarations (#includes and #defines) that are necessary to support the main class type, some message parameters, and some argument values. Aggregate class specifications always include the component classes that make up the class. Each message declaration should also have an associated textual description. Each description should contain complete instructions for proper message use and describe the message parameters and return values. The instructions should explain the function that each message accomplishes and the effects of the message on the rest of the system. The information should be complete enough that you need not reference the implementing code when using the class. You can often place this information directly into a developers guide or manual.

Inheritance is most beneficial when you can exploit polymorphism. Because the language supports neither use of these features, using inheritance in C is a limited and manual process. However, the effort may still be worthwhile to represent the design of the class hierarchy in the code.

Subclass specifications require explicit use of macros to rename inherited superclass messages. The new subclass type is often called the typedef'ed superclass's main type. The superclass's type could also become a component of the new subclass structure, which may also define its own additional or overriding attributes. You declare adjunct subclass messages as previously described. Because it is an inheritance relationship, direct access of superclass instance-attribute data from within the subclass methods is appropriate, but maintaining private treatment is usually better in the long term.

At best, inheritance without polymorphism provides only some organizational benefits that you can often realize anyway using aggregation associations from the outset. Weigh the perceived benefits against the required effort before proceeding with an inheritance implementation using C.

IMPLEMENTING CLASS METHODS

Generally, each class message declared in the header file corresponds with method that has a matching signature in the associated definition file. This file also contains localized type declarations, #defines and #includes, and any class-attribute data and private-method definitions required to maintain the overall class state and to support the implementations of the public-class methods. In every case, the file includes its corresponding class-declaration header file.

Both private and public methods can freely and directly use class attributes, but only class messages should allow external access or modification. The attributes provide data values that are common to and vary uniformly across all instances that the class produces. Class attributes may track simple class states or may completely define the structure of a singular class instance.

Private methods are statically declared C-function definitions. You invoke them from within the public methods. This process is internal message sending. They help to keep the public methods smaller and logically efficient by factoring out any repeated or complex code sequences into single-function operations. Because the methods are private to the file, you declare them without a prototype. The function definition serves as the internal-message signature. No class-name prefix is required, but you should use a "self" parameter whenever you pass instance variables to maintain convention.

Private methods can send messages to other objects, use class attributes to form attribute values for specific instances, and call the class's own public methods whenever appropriate. They should not manipulate data external to the class definition. Each private method should have an associated textual description documenting the algorithmic execution of the operation, including inputs and outputs. **Listing 4** shows the class attributes and methods in a partial definition file for the pipe class.

A class's public methods are the complete implementations of its corresponding messages. By following the same naming conventions, each public method's signature matches a corresponding message prototype. If you use macros to directly implement some messages, no corresponding method will ex-

LISTING 4—LOCAL-CLASS ATTRIBUTES AND METHODS

Class Specification

```
    #include "pipe.h"
...
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
...
/*
 * Class attribute descriptions...
 */
static const char    *pipeName = "/dev/pipe";
static LongInteger  nextId       = 0;
...
...
/*
 * Algorithmic description...
 */
static void
formNextName (Pipe_Pointer self)
{
...
   self->name = (char *)malloc (sizeof (pipeName)+
      sizeof (nextId) );

   sprintf (self->name, "%s%d", pipeName, nextId);

    ++nextId;
...
}
...
...
```

Uses Class Attributes to Form an Instance Attribute

(continued, pipe.c)

ist in the definition file for those messages. Limit the use of macro implementations to simple attribute modification or retrieval that requires no interdependent manipulation or to situations in which execution efficiency is a concern.

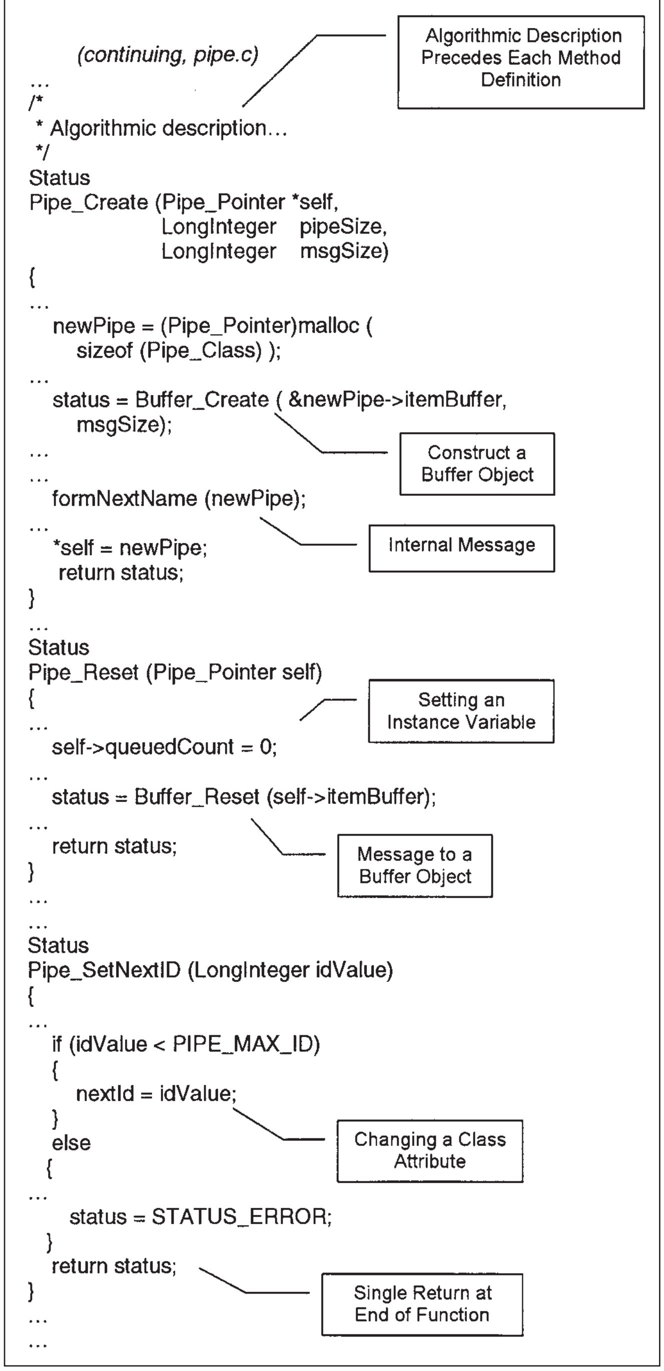
Public-instance methods carry out the major behaviors of the class's created objects. Each method may declare objects of other classes, send messages to new objects and component objects, and invoke private or other public methods from its class. They should neither provide external access to class-attribute values nor directly manipulate data outside the class definition. As with private methods, each public-instance method should have an associated algorithmic description stating how you perform the operation. It should also describe the method's parameters and return values.

Instance methods should be concise. Operations should have only one exit whenever possible. Conforming to this convention helps reduce surprises in runtime execution when debugging or enhancing code. It is also easier for developers to work on components designed by others when all the components follow a similar form.

The same guidelines govern public-class methods and instance methods. The only variance is that public-class methods affect the whole class, by operating solely on class-attribute data instead of instance variables.

Listing 5 illustrates partial method implementations for the pipe-class definition.

LISTING 5—PUBLIC METHODS


Algorithmic Description Precedes Each Method Definition
Construct a Buffer Object
Internal Message
Setting an Instance Variable
Message to a Buffer Object
Changing a Class Attribute
Single Return at End of Function

```
        (continuing, pipe.c)
...
/*
 * Algorithmic description...
 */
Status
Pipe_Create (Pipe_Pointer *self,
             LongInteger   pipeSize,
             LongInteger   msgSize)
{
...
  newPipe = (Pipe_Pointer)malloc (
    sizeof (Pipe_Class) );
...
  status = Buffer_Create ( &newPipe->itemBuffer,
    msgSize);
...
...
  formNextName (newPipe);
...
  *self = newPipe;
   return status;
}
...
Status
Pipe_Reset (Pipe_Pointer self)
{
...
  self->queuedCount = 0;
...
  status = Buffer_Reset (self->itemBuffer);
...
  return status;
}
...
...
Status
Pipe_SetNextID (LongInteger idValue)
{
...
  if (idValue < PIPE_MAX_ID)
  {
    nextId = idValue;
  }
  else
  {
...
    status = STATUS_ERROR;
  }
  return status;
}
...
...
```

A CLASS TEMPLATE FOR C

Once you understand OOD components and how they are mapped into ANSI C, you can easily distill a template for organizing C constructs into classes. Even if you do not religiously follow these constructs, a template can help produce more uniform, consistent, and conceptually organized code.

The class-specification layout in a C header file generally conforms to the following sequence:

1. All #includes for component and associated classes that are required to declare the class type and message parameters;
2. Ancillary #defines, constants, and types needed to support the class types and message arguments;
3. The typedef'ed main class type declaration containing the instance attribute declarations;
4. The typedef'ed pointer type referring to the main class type if necessary;
5. The function prototype declarations for the object constructors and destructors;
6. The function prototype declarations or macro definitions for all instance messages;
7. The function prototype declarations or macro definitions for all class messages.

For singular class instances, omit the class type, the class-type reference, and all instance messages.

The class-implementation layout in a C definition file generally conforms to the following sequence:

1. The *#include* of the class's public specification header file;
2. All *#includes* for component and associated classes that are required locally to declare the class attributes and to support method algorithms;
3. Ancillary *#defines*, constants, and types needed to support the class attributes and method algorithms;
4. Class-attribute declarations, if any;
5. Local internal method definitions, if any;
6. Public-instance-method definitions for each instance message that a

macro does not already define;

7. Public-class-method definitions for each class message that a macro does not already define.

For singular class instances, omit all instance methods and use class attributes.

Although a true OOP language best serves OODs, you can still effectively use the methodology with a conventional procedural language, such as ANSI C. You can easily master the straightforward coding conventions and bring many of the benefits of OOD to projects that you implement in standard C.

By following these conventions, programs and their component objects become more robust, changeable, and reusable. Coding standards based on these guidelines encourage uniform implementation of object-based designs, regardless of language, and help promote the correct use of the OOD methodology. They bring the ultimate goal of better software systems for everyone a few steps closer.□

References

1. Booch, Grady, *Object oriented design with applications*, The Benjamin/Cummings Publishing Co, 1991.

2. EVB Software Engineering Inc, "Object oriented development for Ada software", 1989.

3. Kernighan, Brian W, and Dennis M Richie, *The C Programming Language*, Second Edition, Prentice Hall, 1988.

4. Lippman, Stanley B, *C++ Primer*, Second Edition, Addison Wesley, 1991.

5. Rumbaugh, James, Michael Blaha, et al, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

6. Sphar, Chuck, *Object-Oriented Programming Power*, Microsoft Press, 1991.

7. Wirfs-Brock, Rebecca, Brian Wilkerson, and Lauren Wiener, *Designing object-oriented software*, Prentice Hall, 1990.

Author's biography

Edward Binder is a software staff engineer at Motorola Computer Group, where he has worked for 15 years. He designs and implements validation-test software for single-board-computer products and has worked on Motorola Power PC embedded-computer products. He has a BSEE from Arizona State University (Tempe) and enjoys astronomy, photography, and hiking.

Circle 3 or visit www.ednmag.com/infoaccess.asp