Classes and Objects Chapter Ten

10.1 Chapter Overview

Many modern imperative high level languages support the notion of classes and objects. C++ (an object version of C), Java, and Delphi (an object version of Pascal) are two good examples. Of course, these high level language compilers translate their high level source code into low-level machine code, so it should be pretty obvious that some mechanism exists in machine code for implementing classes and objects.

Although it has always been possible to implement classes and objects in machine code, most assemblers provide poor support for writing object-oriented assembly language programs. Of course, HLA does not suffer from this drawback as it provides good support for writing object-oriented assembly language programs. This chapter discusses the general principles behind object-oriented programming (OOP) and how HLA supports OOP.

10.2 General Principles

Before discussing the mechanisms behind OOP, it is probably a good idea to take a step back and explore the benefits of using OOP (especially in assembly language programs). Most texts describing the benefits of OOP will mention buzz-words like “code reuse,” “abstract data types,” “improved development efficiency,” and so on. While all of these features are nice and are good attributes for a programming paradigm, a good software engineer would question the use of assembly language in an environment where “improved development efficiency” is an important goal. After all, you can probably obtain far better efficiency by using a high level language (even in a non-OOP fashion) than you can by using objects in assembly language. If the purported features of OOP don’t seem to apply to assembly language programming, why bother using OOP in assembly? This section will explore some of those reasons.

The first thing you should realize is that the use of assembly language does not negate the aforementioned OOP benefits. OOP in assembly language does promote code reuse, it provides a good method for implementing abstract data types, and it can improve development efficiency *in assembly language*. In other words, if you’re dead set on using assembly language, there are benefits to using OOP.

To understand one of the principle benefits of OOP, consider the concept of a global variable. Most programming texts strongly recommend against the use of global variables in a program (as does this text). Interprocedural communication through global variables is dangerous because it is difficult to keep track of all the possible places in a large program that modify a given global object. Worse, it is very easy when making enhancements to accidentally reuse a global object for something other than its intended purpose; this tends to introduce defects into the system.

Despite the well-understood problems with global variables, the semantics of global objects (extended lifetimes and accessibility from different procedures) are absolutely necessary in various situations. Objects solve this problem by letting the programmer decide on the lifetime of an object[1] as well as allow access to data fields from different procedures. Objects have several advantages over simple global variables insofar as objects can control access to their data fields (making it difficult for procedures to accidentally access the data) and you can also create multiple *instances* of an object allowing two separate sections of your program to use their own unique “global” object without interference from the other section.

Of course, objects have many other valuable attributes. One could write several volumes on the benefits of objects and OOP; this single chapter cannot do this subject justice. The following subsections present objects with an eye towards using them in HLA/assembly programs. However, if you are a beginning to

1. That is, the time during which the system allocates memory for an object.

OOP or wish more information about the object-oriented paradigm, you should consult other texts on this subject.

An important use for classes and objects is to create *abstract data types* (ADTs). An abstract data type is a collection of data objects and the functions (which we'll call *methods*) that operate on the data. In a pure abstract data type, the ADT's methods are the only code that has access to the data fields of the ADT; external code may only access the data using function calls to get or set data field values (these are the ADT's *accessor* methods). In real life, for efficiency reasons, most languages that support ADTs allow, at least, limited access to the data fields of an ADT by external code.

Assembly language is not a language most people associate with ADTs. Nevertheless, HLA provides several features to allow the creation of rudimentary ADTs. While some might argue that HLA's facilities are not as complete as those in a language such as C++ or Java, keep in mind that these differences exist because HLA is assembly language.

True ADTs should support *information hiding*. This means that the ADT does not allow the user of an ADT access to internal data structures and routines which manipulate those structures. In essence, information hiding restricts access to an ADT to only the accessor methods provided by the ADT. Assembly language, of course, provides very few restrictions. If you are dead set on accessing an object directly, there is very little HLA can do to prevent you from doing this. However, HLA has some facilities which will provide a small amount of information hiding capabilities. Combined with some care on your part, you will be able to enjoy many of the benefits of information hiding within your programs.

The primary facility HLA provides to support information hiding is separate compilation, linkable modules, and the #INCLUDE/#INCLUDEONCE directives. For our purposes, an abstract data type definition will consist of two sections: an *interface* section and an *implementation* section.

The interface section contains the definitions which must be visible to the application program. In general, it should not contain any specific information which would allow the application program to violate the information hiding principle, but this is often impossible given the nature of assembly language. Nevertheless, you should attempt to only reveal what is absolutely necessary within the interface section.

The implementation section contains the code, data structures, etc., to actually implement the ADT. While some of the methods and data types appearing in the implementation section may be public (by virtue of appearance within the interface section), many of the subroutines, data items, and so on will be private to the implementation code. The implementation section is where you hide all the details from the application program.

If you wish to modify the abstract data type at some point in the future, you will only have to change the interface and implementation sections. Unless you delete some previously visible object which the applications use, there will be no need to modify the applications at all.

Although you could place the interface and implementation sections directly in an application program, this would not promote information hiding or maintainability, especially if you have to include the code in several different applications. The best approach is to place the implementation section in an include file that any interested application reads using the HLA #INCLUDE directive and to place the implementation section in a separate module that you link with your applications.

The include file would contain EXTERNAL directives, any necessary macros, and other definitions you want made public. It generally would not contain 80x86 code except, perhaps, in some macros. When an application wants to make use of an ADT it would include this file.

The separate assembly file containing the implementation section would contain all the procedures, functions, data objects, etc., to actually implement the ADT. Those names which you want to be public should appear in the interface include file and have the EXTERNAL attribute. You should also include the interface include file in the implementation file so you do not have to maintain two sets of EXTERNAL directives.

One problem with using procedures for data access methods is the fact that many accessor methods are especially trivial (typically just a MOV instruction) and the overhead of the call and return instructions is expensive for such trivial operations. For example, suppose you have an ADT whose data object is a structure, but you do not want to make the field names visible to the application and you really do not want to allow the application to access the fields of the data structure directly (because the data structure may change in the future). The normal way to handle this is to supply a method *GetField* which returns the desired field of the object. However, as pointed out above, this can be very slow. An alternative, for simple access methods is to use a macro to emit the code to access the desired field. Although code to directly access the data object appears in the application program (via macro expansion), it will be automatically updated if you ever change the macro in the interface section by simply assembling your application.

Although it is quite possible to create ADTs using nothing more than separate compilation and, perhaps, RECORDs, HLA does provide a better solution: the class. Read on to find out about HLA's support for classes and objects as well as how to use these to create ADTs.

10.3 Classes in HLA

HLA's classes provide a good mechanism for creating abstract data types. Fundamentally, a class is little more than a RECORD declaration that allows the definition of fields other than data fields (e.g., procedures, constants, and macros). The inclusion of other program declaration objects in the class definition dramatically expands the capabilities of a class over that of a record. For example, with a class it is now possible to easily define an ADT since classes may include data and methods that operate on that data (procedures).

The principle way to create an abstract data type in HLA is to declare a class data type. Classes in HLA always appear in the TYPE section and use the following syntax:

```
classname :   class

                  << Class declaration section >>

              endclass;
```

The class declaration section is very similar to the local declaration section for a procedure insofar as it allows CONST, VAL, VAR, and STATIC variable declaration sections. Classes also let you define macros and specify procedure, iterator, and *method* prototypes (method declarations are legal only in classes). Conspicuously absent from this list is the TYPE declaration section. You cannot declare new types within a class.

A method is a special type of procedure that appears only within a class. A little later you will see the difference between procedures and methods, for now you can treat them as being one and the same. Other than a few subtle details regarding class initialization and the use of pointers to classes, their semantics are identical[2]. Generally, if you don't know whether to use a procedure or method in a class, the safest bet is to use a method.

You do not place procedure/iterator/method code within a class. Instead you simply supply *prototypes* for these routines. A routine prototype consists of the PROCEDURE, ITERATOR, or METHOD reserved word, the routine name, any parameters, and a couple of optional procedure attributes (@USE, RETURNS, and EXTERNAL). The actual routine definition (i.e., the body of the routine and any local declarations it needs) appears outside the class.

The following example demonstrates a typical class declaration appearing in the TYPE section:

```
TYPE
    TypicalClass: class

        const
            TCconst := 5;

        val
```

2. Note, however, that the difference between procedures and methods makes all the difference in the world to the object-oriented programming paradigm. Hence the inclusion of methods in HLA's class definitions.

```
        TCval := 6;

    var
        TCvar : uns32;        // Private field used only by TCproc.

    static
        TCstatic : int32;

    procedure TCproc( u:uns32 ); returns( "eax" );
    iterator TCiter( i:int32 ); external;
    method TCmethod( c:char );

endclass;
```

As you can see, classes are very similar to records in HLA. Indeed, you can think of a record as being a class that only allows VAR declarations. HLA implements classes in a fashion quite similar to records insofar as it allocates sequential data fields in sequential memory locations. In fact, with only one minor exception, there is almost no difference between a RECORD declaration and a CLASS declaration that only has a VAR declaration section. Later you'll see exactly how HLA implements classes, but for now you can assume that HLA implements them the same as it does records and you won't be too far off the mark.

You can access the *TCvar* and *TCstatic* fields (in the class above) just like a record's fields. You access the CONST and VAL fields in a similar manner. If a variable of type *TypicalClass* has the name *obj*, you can access the fields of *obj* as follows:

```
mov ( obj.TCconst, eax );
mov( obj.TCval, ebx );
add( obj.TCvar, eax );
add( obj.TCstatic, ebx );
obj.TCproc( 20 );        // Calls the TCproc procedure in TypicalClass.
etc.
```

If an application program includes the class declaration above, it can create variables using the *TypicalClass* type and perform operations using the above methods. Unfortunately, the application program can also access the fields of the *ADT* data type with impunity. For example, if a program created a variable *MyClass* of type *TypicalClass*, then it could easily execute instructions like "MOV(MyClass.TCvar, eax);" even though this field might be private to the implementation section. Unfortunately, if you are going to allow an application to declare a variable of type *TypicalClass*, the field names will have to be visible. While there are some tricks we could play with HLA's class definitions to help hide the private fields, the best solution is to thoroughly comment the private fields and then exercise some restraint when accessing the fields of that class. Specifically, this means that ADTs you create using HLA's classes cannot be "pure" ADTs since HLA allows direct access to the data fields. However, with a little discipline, you can simulate a pure ADT by simply electing not to access such fields outside the class' methods, procedures, and iterators.

Prototypes appearing in a class are effectively FORWARD declarations. Like normal forward declarations, all procedures, iterators, and methods you define in a class must have an actual implementation later in the code. Alternately, you may attach the EXTERNAL keyword to the end of a procedure, iterator, or method declaration within a class to inform HLA that the actual code appears in a separate module. As a general rule, class declarations appear in header files and represent the interface section of an ADT. The procedure, iterator, and method bodies appear in the implementation section which is usually a separate source file that you compile separately and link with the modules that use the class.

The following is an example of a sample class procedure implementation:

```
procedure TypicalClass.TCproc( u:uns32 ); nodisplay;
    << Local declarations for this procedure >>
begin TCproc;

    << Code to implement whatever this procedure does >>

end TCProc;
```


There are several differences between a standard procedure declaration and a class procedure declaration. First, and most obvious, the procedure name includes the class name (e.g., *TypicalClass.TCproc*). This differentiates this class procedure definition from a regular procedure that just happens to have the name *TCproc*. Note, however, that you do not have to repeat the class name before the procedure name in the BEGIN and END clauses of the procedure (this is similar to procedures you define in HLA NAMESPACEs).

A second difference between class procedures and non-class procedures is not obvious. Some procedure attributes (@USE, EXTERNAL, RETURNS, @CDECL, @PASCAL, and @STDCALL) are legal only in the prototype declaration appearing within the class while other attributes (@NOFRAME, @NODISPLAY, @NOALIGNSTACK, and ALIGN) are legal only within the procedure definition and not within the class. Fortunately, HLA provides helpful error messages if you stick the option in the wrong place, so you don't have to memorize this rule.

If a class routine's prototype does not have the EXTERNAL option, the compilation unit (that is, the PROGRAM or UNIT) containing the class declaration must also contain the routine's definition or HLA will generate an error at the end of the compilation. For small, local, classes (i.e., when you're embedding the class declaration and routine definitions in the same compilation unit) the convention is to place the class' procedure, iterator, and method definitions in the source file shortly after the class declaration. For larger systems (i.e., when separately compiling a class' routines), the convention is to place the class declaration in a header file by itself and place all the procedure, iterator, and method definitions in a separate HLA unit and compile them by themselves.

10.4 Objects

Remember, a class definition is just a type. Therefore, when you declare a class type you haven't created a variable whose fields you can manipulate. An *object* is an *instance* of a class; that is, an object is a variable that is some class type. You declare objects (i.e., class variables) the same way you declare other variables: in a VAR, STATIC, or STORAGE section[3]. A pair of sample object declarations follow:

```
var
    T1: TypicalClass;
    T2: TypicalClass;
```

For a given class object, HLA allocates storage for each variable appearing in the VAR section of the class declaration. If you have two objects, *T1* and *T2*, of type *TypicalClass* then *T1.TCvar* is unique as is *T2.TCvar*. This is the intuitive result (similar to RECORD declarations); most data fields you define in a class will appear in the VAR declaration section.

Static data objects (e.g., those you declare in the STATIC section of a class declaration) are not unique among the objects of that class; that is, HLA allocates only a single static variable that all variables of that class share. For example, consider the following (partial) class declaration and object declarations:

```
type
    sc: class

        var
            i:int32;

        static
            s:int32;
            .
            .
            .
    endclass;

var
```

3. Technically, you could also declare an object in a READONLY section, but HLA does not allow you to define class constants, so there is little utility in declaring class objects in the READONLY section.

```
s1: sc;
s2: sc;
```

In this example, *s1.i* and *s2.i* are different variables. However, *s1.s* and *s2.s* are aliases of one another Therefore, an instruction like "mov(5, s1.s);" also stores five into *s2.s*. Generally you use static class variables to maintain information about the whole class while you use class VAR objects to maintain information about the specific object. Since keeping track of class information is relatively rare, you will probably declare most class data fields in a VAR section.

You can also create dynamic instances of a class and refer to those dynamic objects via pointers. In fact, this is probably the most common form of object storage and access. The following code shows how to create pointers to objects and how you can dynamically allocate storage for an object:

```
var
    pSC: pointer to sc;
        .
        .
        .
    malloc( @size( sc ) );
    mov( eax, pSC );
        .
        .
        .
    mov( pSC, ebx );
    mov( (type sc [ebx]).i, eax );
```

Note the use of type coercion to cast the pointer in EBX as type *sc*.

10.5 Inheritance

Inheritance is one of the most fundamental ideas behind object-oriented programming. The basic idea behind inheritance is that a class inherits, or copies, all the fields from some class and then possibly expands the number of fields in the new data type. For example, suppose you created a data type *point* which describes a point in the planar (two dimensional) space. The class for this point might look like the following:

```
type
    point: class

        var
            x:int32;
            y:int32;

        method distance;

    endclass;
```

Suppose you want to create a point in 3D space rather than 2D space. You can easily build such a data type as follows:

```
type
    point3D: class inherits( point );

        var
            z:int32;

    endclass;
```


The INHERITS option on the CLASS declaration tells HLA to insert the fields of *point* at the beginning of the class. In this case, *point3D* inherits the fields of *point*. HLA always places the inherited fields at the beginning of a class object. The reason for this will become clear a little later. If you have an instance of *point3D* which you call *P3*, then the following 80x86 instructions are all legal:

```
mov( P3.x, eax );
add( P3.y, eax );
mov( eax, P3.z );
P3.distance();
```

Note that the *P3.distance* method invocation in this example calls the *point.distance* method. You do not have to write a separate *distance* method for the *point3D* class unless you really want to do so (see the next section for details). Just like the *x* and *y* fields, *point3D* objects inherit *point's* methods.

10.6 Overriding

Overriding is the process of replacing an existing method in an inherited class with one more suitable for the new class. In the *point* and *point3D* examples appearing in the previous section, the *distance* method (presumably) computes the distance from the origin to the specified point. For a point on a two-dimensional plane, you can compute the distance using the function:

$$dist = \sqrt{x^2+y^2}$$

However, the distance for a point in 3D space is given by the equation:

$$dist = \sqrt{x^2+y^2+z^2}$$

Clearly, if you call the *distance* function for *point* for a *point3D* object you will get an incorrect answer. In the previous section, however, you saw that the *P3* object calls the distance function inherited from the *point* class. Therefore, this would produce an incorrect result.

In this situation the *point3D* data type must override the *distance* method with one that computes the correct value. You cannot simply redefine the *point3D* class by adding a *distance* method prototype:

```
type
    point3D:   class inherits( point )

        var
            z:int32;

        method distance;   // This doesn't work!

    endclass;
```

The problem with the *distance* method declaration above is that *point3D* already has a distance method – the one that it inherits from the *point* class. HLA will complain because it doesn't like two methods with the same name in a single class.

To solve this problem, we need some mechanism by which we can override the declaration of *point.distance* and replace it with a declaration for *point3D.distance*. To do this, you use the OVERRIDE keyword before the method declaration:

```
type
    point3D:   class inherits( point )

        var
            z:int32;

        override method distance;   // This will work!

    endclass;
```

The OVERRIDE prefix tells HLA to ignore the fact that *point3D* inherits a method named distance from the *point* class. Now, any call to the *distance* method via a *point3D* object will call the *point3D.distance* method rather than *point.distance*. Of course, once you override a method using the OVERRIDE prefix, you must supply the method in the implementation section of your code, e.g.,

```
method point3D.distance; nodisplay;

    << local declarations for the distance function>>

begin distance;

    << Code to implement the distance function >>

end distance;
```

10.7 Virtual Methods vs. Static Procedures

A little earlier, this chapter suggested that you could treat class methods and class procedures the same. There are, in fact, some major differences between the two (after all, why have methods if they're the same as procedures?). As it turns out, the differences between methods and procedures is crucial if you want to develop object-oriented programs. Methods provide the second feature necessary to support true polymorphism: virtual procedure calls[4]. A virtual procedure call is just a fancy name for an indirect procedure call (using a pointer associated with the object). The key benefit of virtual procedures is that the system automatically calls the right method when using pointers to generic objects.

Consider the following declarations using the *point* class from the previous sections:

```
var
    P2: point;
    P: pointer to point;
```

Given the declarations above, the following assembly statements are all legal:

```
    mov( P2.x, eax );
    mov( P2.y, ecx );
    P2.distance();          // Calls point3D.distance.

    lea( ebx, P2 );         // Store address of P2 into P.
    mov( ebx, P );
    P.distance();           // Calls point.distance.
```

Note that HLA lets you call a method via a pointer to an object rather than directly via an object variable. This is a crucial feature of objects in HLA and a key to implementing *virtual method calls*.

The magic behind polymorphism and inheritance is that object pointers are *generic*. In general, when your program references data indirectly through a pointer, the value of the pointer should be the address of the underlying data type associated with that pointer. For example, if you have a pointer to a 16-bit unsigned integer, you wouldn't normally use that pointer to access a 32-bit signed integer value. Similarly, if you have a pointer to some record, you would not normally cast that pointer to some other record type and access the fields of that other type[5]. With pointers to class objects, however, we can lift this restriction a bit. Pointers to objects may legally contain the address of the object's type *or the address of any object that inherits the fields of that type*. Consider the following declarations that use the *point* and *point3D* types from the previous examples:

```
var
```

4. Polymorphism literally means "many-faced." In the context of object-oriented programming polymorphism means that the same method name, e.g., *distance*, and refer to one of several different methods.
5. Of course, assembly language programmers break rules like this all the time. For now, let's assume we're playing by the rules and only access the data using the data type associated with the pointer.

```
P2: point;
P3: point3D;
p: pointer to point;
    .
    .
    .
lea( ebx, P2 );
mov( ebx, p );
p.distance();         // Calls the point.distance method.
    .
    .
    .
lea( ebx, P3 );
mov( ebx, p );        // Yes, this is semantically legal.
p.distance();         // Surprise, this calls point3D.distance.
```

Since *p* is a pointer to a *point* object, it might seem intuitive for *p.distance* to call the *point.distance* method. However, methods are *polymorphic*. If you've got a pointer to an object and you call a method associated with that object, the system will call the actual (overridden) method associated with the object, not the method specifically associated with the pointer's class type.

Class procedures behave differently than methods with respect to overridden procedures. When you call a class procedure indirectly through an object pointer, the system will always call the procedure associated with the underlying class associated with the pointer. So had *distance* been a procedure rather than a method in the previous examples, the "p.distance();" invocation would always call *point.distance*, even if *p* is pointing at a *point3D* object. The section on Object Initialization, later in this chapter, explains why methods and procedures are different (see "Object Implementation" on page 1071).

Note that iterators are also virtual; so like methods an object iterator invocation will always call the (overridden) iterator associated with the actual object whose address the pointer contains. To differentiate the semantics of methods and iterators from procedures, we will refer to the method/iterator calling semantics as *virtual procedures* and the calling semantics of a class procedure as a *static procedure*.

10.8 Writing Class Methods, Iterators, and Procedures

For each class procedure, method, and iterator prototype appearing in a class definition, there must be a corresponding procedure, method, or iterator appearing within the program (for the sake of brevity, this section will use the term *routine* to mean procedure, method, or iterator from this point forward). If the prototype does not contain the EXTERNAL option, then the code must appear in the same compilation unit as the class declaration. If the EXTERNAL option does follow the prototype, then the code may appear in the same compilation unit or a different compilation unit (as long as you link the resulting object file with the code containing the class declaration). Like external (non-class) procedures and iterators, if you fail to provide the code the linker will complain when you attempt to create an executable file. To reduce the size of the following examples, they will all define their routines in the same source file as the class declaration.

HLA class routines must always follow the class declaration in a compilation unit. If you are compiling your routines in a separate unit, the class declarations must still precede the code with the class declaration (usually via an #INCLUDE file). If you haven't defined the class by the time you define a routine like *point.distance*, HLA doesn't know that *point* is a class and, therefore, doesn't know how to handle the routine's definition.

Consider the following declarations for a point2D class:

```
type
    point2D: class

        const
```



```
        UnitDistance: real32 := 1.0;

    var
        x: real32;
        y: real32;

    static
        LastDistance: real32;

    method distance( fromX: real32;  fromY:real32 ); returns( "st0" );
    procedure InitLastDistance;

endclass;
```

The distance function for this class should compute the distance from the object's point to (fromX,fromY). The following formula describes this computation:

$$\sqrt{(x - fromX)^2 + (y - fromY)^2}$$

A first pass at writing the distance method might produce the following code:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( x );          // Note: this doesn't work!
    fld( fromX );      // Compute (x-fromX)
    fsub();
    fld( st0 );        // Duplicate value on TOS.
    fmul();            // Compute square of difference.

    fld( y );          // This doesn't work either.
    fld( fromY );      // Compute (y-fromY)
    fsub();
    fld( st0 );        // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;
```

This code probably looks like it should work to someone who is familiar with an object-oriented programming language like C++ or Delphi. However, as the comments indicate, the instructions that push the *x* and *y* variables onto the FPU stack don't work – HLA doesn't automatically define the symbols associated with the data fields of a class within that class' routines.

To learn how to access the data fields of a class within that class' routines, we need to back up a moment and discover some very important implementation details concerning HLA's classes. To do this, consider the following variable declarations:

```
var
    Origin: point2D;
    PtInSpace: point2D;
```

Remember, whenever you create two objects like *Origin* and *PtInSpace*, HLA reserves storage for the *x* and *y* data fields for both of these objects. However, there is only one copy of the *point2D.distance* method in memory. Therefore, were you to call *Origin.distance* and *PtInSpace.distance*, the system would call the same routine for both method invocations. Once inside that method, one has to wonder what an instruction like "fld(x);" would do. How does it associate *x* with *Origin.x* or *PtInSpace.x*? Worse still, how would this code differentiate between the data field *x* and a global object *x*? In HLA, the answer is "it doesn't." You do not specify the data field names within a class routine by simply using their names as though they were common variables.

To differentiate *Origin.x* from *PtInSpace.x* within class routines, HLA automatically passes a pointer to an object's data fields whenever you call a class routine. Therefore, you can reference the data fields indirectly off this pointer. HLA passes this object pointer in the ESI register. This is one of the few places where HLA-generated code will modify one of the 80x86 registers behind your back: **anytime you call a class routine, HLA automatically loads the ESI register with the object's address**. Obviously, you cannot count on ESI's value being preserved across class routine class nor can you pass parameters to the class routine in the ESI register (though it is perfectly reasonable to specify "@USE ESI;" to allow HLA to use the ESI register when setting up other parameters). For class methods and iterators (but not procedures), HLA will also load the EDI register with the address of the class' *virtual method table* (see "Virtual Method Tables" on page 1073). While the virtual method table address isn't as interesting as the object address, keep in mind that **HLA-generated code will overwrite any value in the EDI register when you call a method or an iterator**. Again, "EDI" is a good choice for the @USE operand for methods since HLA will wipe out the value in EDI anyway.

Upon entry into a class routine, ESI contains a pointer to the (non-static) data fields associated with the class. Therefore, to access fields like *x* and *y* (in our *point2D* example), you could use an address expression like the following:

```
(type point2D [esi].x
```

Since you use ESI as the base address of the object's data fields, it's a good idea not to disturb ESI's value within the class routines (or, at least, preserve ESI's value if you need to access the objects data fields after some point where you must use ESI for some other purpose). Note that if you call an iterator or a method you do not have to preserve EDI (unless, for some reason, you need access to the virtual method table, which is unlikely).

Accessing the fields of a data object within a class' routines is such a common operation that HLA provides a shorthand notation for casting ESI as a pointer to the class object: THIS. Within a class in HLA, the reserved word THIS automatically expands to a string of the form "(type *classname* [esi])" substituting, of course, the appropriate class name for *classname*. Using the THIS keyword, we can (correctly) rewrite the previous distance method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );       // Compute (x-fromX)
    fsub();
    fld( st0 );         // Duplicate value on TOS.
    fmul();             // Compute square of difference.

    fld( this.y );
    fld( fromY );       // Compute (y-fromY)
    fsub();
    fld( st0 );         // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;
```

Don't forget that calling a class routine wipes out the value in the ESI register. This isn't obvious from the syntax of the routine's invocation. It is especially easy to forget this when calling some class routine from inside some other class routine; don't forget that if you do this the internal call wipes out the value in ESI and on return from that call ESI no longer points at the original object. Always push and pop ESI (or otherwise preserve ESI's value) in this situation, e.g.,

```
    .
```

```
.
.
fld( this.x );    // ESI points at current object.
.
.
.
push( esi );                 // Preserve ESI across this method call.
SomeObject.SomeMethod();
pop( esi );
.
.
.
lea( ebx, this.x );          // ESI points at original object here.
```

The THIS keyword provides access to the class variables you declare in the VAR section of a class. You can also use THIS to call other class routines associated with the current object, e.g.,

```
this.distance( 5.0, 6.0 );
```

To access class constants and STATIC data fields you generally do not use the THIS pointer. HLA associates constant and static data fields with the whole class, not a specific object. To access these class members, just use the class name in place of the object name. For example, to access the *UnitDistance* constant in the *point2D* class you could use a statement like the following:

```
fld( point2D.UnitDistance );
```

As another example, if you wanted to update the *LastDistance* field in the *point2D* class each time you computed a distance, you could rewrite the *point2D.distance* method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );     // Compute (x-fromX)
    fsub();
    fld( st0 );       // Duplicate value on TOS.
    fmul();           // Compute square of difference.

    fld( this.y );
    fld( fromY );     // Compute (y-fromY)
    fsub();
    fld( st0 );       // Compute the square of the difference.
    fmul();

    fsqrt();

    fst( point2D.LastDistance );    // Update shared (STATIC) field.

end distance;
```

To understand why you use the class name when referring to constants and static objects but you use THIS to access VAR objects, check out the next section.

Class procedures are also static objects, so it is possible to call a class procedure by specifying the class name rather than an object name in the procedure invocation, e.g., both of the following are legal:

```
Origin.InitLastDistance();
point2D.InitLastDistance();
```

There is, however, a subtle difference between these two class procedure calls. The first call above loads ESI with the address of the *Origin* object prior to actually calling the *InitLastDistance* procedure. The second call, however, is a direct call to the class procedure without referencing an object; therefore, HLA doesn't know what object address to load into the ESI register. In this case, HLA loads NULL (zero) into ESI prior to calling the *InitLastDistance* procedure. Because you can call class procedures in this manner, it's always a good idea to check the value in ESI within your class procedures to verify that HLA contains an object address. Checking the value in ESI is a good way to determine which calling mechanism is in use. Later, this chapter will discuss constructors and object initialization; there you will see a good use for static procedures and calling those procedures directly (rather than through the use of an object).

10.9 Object Implementation

In a high level object-oriented language like C++ or Delphi, it is quite possible to master the use of objects without really understanding how the machine implements them. One of the reasons for learning assembly language programming is to fully comprehend low-level implementation details so one can make educated decisions concerning the use of programming constructs like objects. Further, since assembly language allows you to poke around with data structures at a very low-level, knowing how HLA implements objects can help you create certain algorithms that would not be possible without a detailed knowledge of object implementation. Therefore, this section, and its corresponding subsections, explains the low-level implementation details you will need to know in order to write object-oriented HLA programs.

HLA implements objects in a manner quite similar to records. In particular, HLA allocates storage for all VAR objects in a class in a sequential fashion, just like records. Indeed, if a class consists of only VAR data fields, the memory representation of that class is nearly identical to that of a corresponding RECORD declaration. Consider the Student record declaration taken from Volume Three and the corresponding class:

```
type
    student:    record
                    Name: char[65];
                    Major: int16;
                    SSN:   char[12];
                    Midterm1: int16;
                    Midterm2: int16;
                    Final: int16;
                    Homework: int16;
                    Projects: int16;
                endrecord;


    student2:   class
                    Name: char[65];
                    Major: int16;
                    SSN:   char[12];
                    Midterm1: int16;
                    Midterm2: int16;
                    Final: int16;
                    Homework: int16;
                    Projects: int16;
                endclass;
```


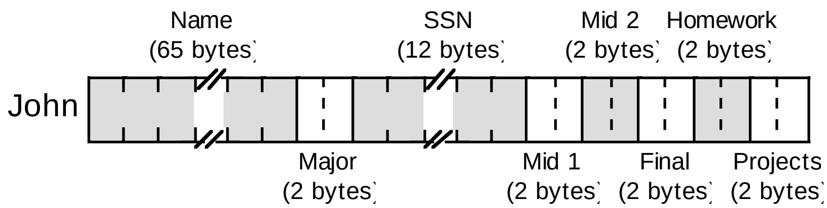

Figure 10.1 Student RECORD Implementation in Memory

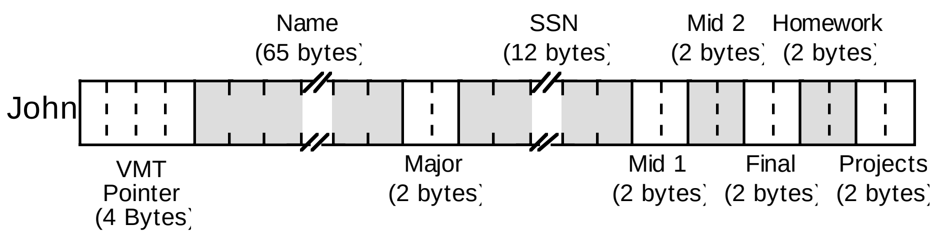

Figure 10.2 Student CLASS Implementation in Memory

If you look carefully at these two figures, you'll discover that the only difference between the class and the record implementations is the inclusion of the VMT (virtual method table) pointer field at the beginning of the class object. This field, which is always present in a class, contains the address of the class' virtual method table which, in turn, contains the addresses of all the class' methods and iterators. The VMT field, by the way, is present even if a class doesn't contain any methods or iterators.

As pointed out in previous sections, HLA does not allocate storage for STATIC objects within the object's storage. Instead, HLA allocates a single instance of each static data field that all objects share. As an example, consider the following class and object declarations:

```
type
    tHasStatic: class

        var
            i:int32;
            j:int32;
            r:real32;

        static
            c:char[2];
            b:byte;

    endclass;

var
    hs1: tHasStatic;
    hs2: tHasStatic;
```

Figure 10.3 shows the storage allocation for these two objects in memory.

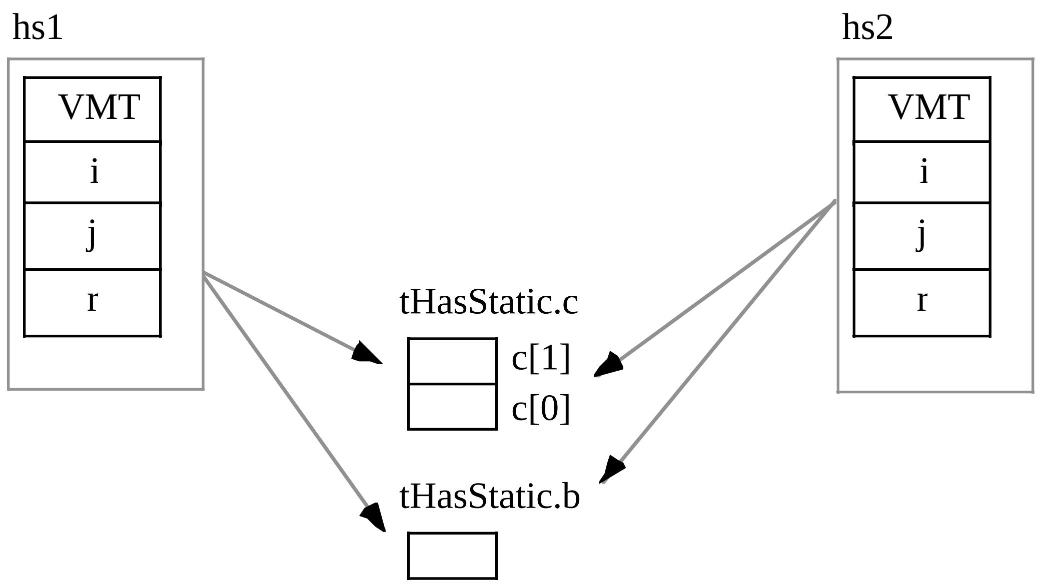

Figure 10.3 Object Allocation with Static Data Fields

Of course, CONST, VAL, and #MACRO objects do not have any run-time memory requirements associated with them, so HLA does not allocate any storage for these fields. Like the STATIC data fields, you may access CONST, VAL, and #MACRO fields using the class name as well as an object name. Hence, even if *tHasStatic* has these types of fields, the memory organization for *tHasStatic* objects would still be the same as shown in Figure 10.3.

Other than the presence of the virtual method table pointer (VMT), the presence of methods, iterators, and procedures has no impact on the storage allocation of an object. Of course, the machine instructions associated with these routines does appear somewhere in memory. So in a sense the code for the routines is quite similar to static data fields insofar as all the objects share a single instance of the routine.

10.9.1 Virtual Method Tables

When HLA calls a class procedure, it directly calls that procedure using a CALL instruction, just like any normal non-class procedure call. Methods and iterators are another story altogether. Each object in the system carries a pointer to a virtual method table which is an array of pointers to all the methods and iterators appearing within the object's class.

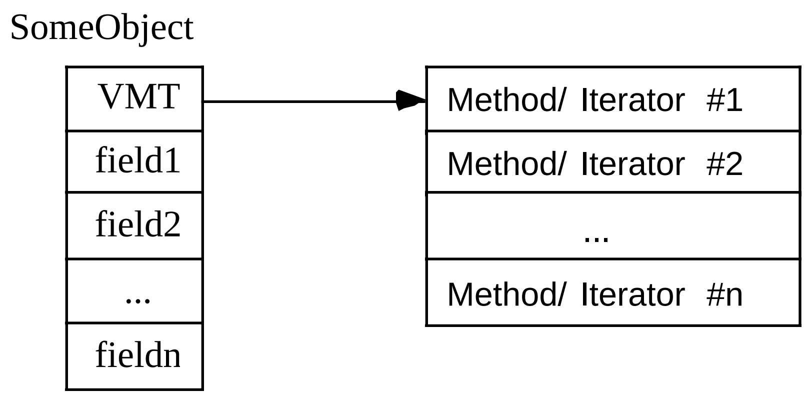

Figure 10.4 Virtual Method Table Organization

Each iterator or method you declare in a class has a corresponding entry in the virtual method table. That dword entry contains the address of the first instruction of that iterator or method. To call a class method or iterator is a bit more work than calling a class procedure (it requires one additional instruction plus the use of the EDI register). Here is a typical calling sequence for a method:

```
mov( ObjectAdrs, ESI );          // All class routines do this.
mov( [esi], edi );               // Get the address of the VMT into EDI
call( (type dword [edi+n]));     // "n" is the offset of the method's entry
                                 //  in the VMT.
```

For a given class there is only one copy of the VMT in memory. This is a static object so all objects of a given class type share the same VMT. This is reasonable since all objects of the same class type have exactly the same methods and iterators (see Figure 10.5).

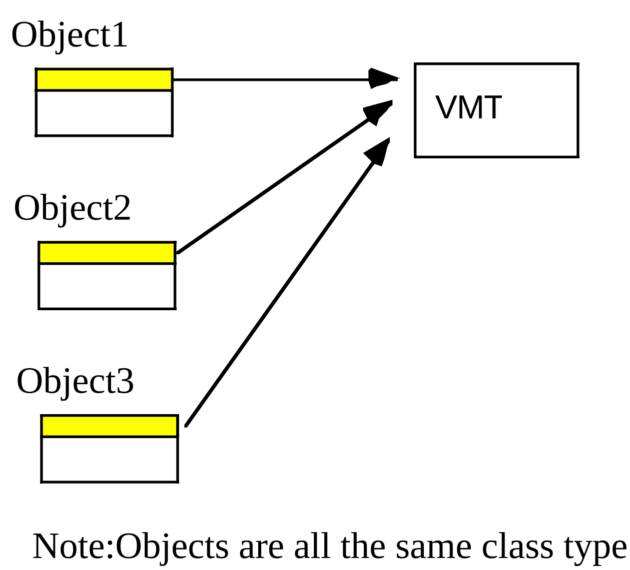

Figure 10.5 All Objects That are the Same Class Type Share the Same VMT

Although HLA builds the VMT record structure as it encounters methods and iterators within a class, HLA does not automatically create the actual run-time virtual method table for you. You must explicitly declare this table in your program. To do this, you include a statement like the following in a STATIC or READONLY declaration section of your program, e.g.,

```
readonly
    VMT( classname );
```

Since the addresses in a virtual method table should never change during program execution, the READONLY section is probably the best choice for declaring VMTs. It should go without saying that changing the pointers in a VMT is, in general, a really bad idea. So putting VMTs in a STATIC section is usually not a good idea.

A declaration like the one above defines the variable *classname._VMT_*. In section 10.10 (see "Constructors and Object Initialization" on page 1079) you see that you'll need this name when initializing object variables. The class declaration automatically defines the *classname._VMT_* symbol as an external static variable. The declaration above just provides the actual definition of this external symbol.

The declaration of a VMT uses a somewhat strange syntax because you aren't actually declaring a new symbol with this declaration, you're simply supplying the data for a symbol that you previously declared implicitly by defining a class. That is, the class declaration defines the static table variable *classname._VMT_*, all you're doing with the VMT declaration is telling HLA to emit the actual data for the table. If, for some reason, you would like to refer to this table using a name other than *classname._VMT_*, HLA does allow you to prefix the declaration above with a variable name, e.g.,

```
readonly
    myVMT: VMT( classname );
```

In this declaration, *myVMT* is an alias of *classname._VMT_*. As a general rule, you should avoid aliases in a program because they make the program more difficult to read and understand. Therefore, it is unlikely that you would ever really need to use this type of declaration.

Like any other global static variable, there should be only one instance of a VMT for a given class in a program. The best place to put the VMT declaration is in the same source file as the class' method, iterator, and procedure code (assuming they all appear in a single file). This way you will automatically link in the VMT whenever you link in the routines for a given class.

10.9.2 Object Representation with Inheritance

Up to this point, the discussion of the implementation of class objects has ignored the possibility of inheritance. Inheritance only affects the memory representation of an object by adding fields that are not explicitly stated in the class declaration.

Adding inherited fields from a *base class* to another class must be done carefully. Remember, an important attribute of a class that inherits fields from a base class is that you can use a pointer to the base class to access the inherited fields from that base class in another class. As an example, consider the following classes:

```
type
    tBaseClass: class
        var
            i:uns32;
            j:uns32;
            r:real32;

        method mBase;
    endclass;

    tChildClassA: class inherits( tBaseClass );
        var
            c:char;
            b:boolean;
```

```
        w:word;

    method mA;
endclass;

tChildClassB: class inherits( tBaseClass );
    var
        d:dword;
        c:char;
        a:byte[3];
endclass;
```

Since both *tChildClassA* and *tChildClassB* inherit the fields of *tBaseClass*, these two child classes include the *i*, *j*, and *r* fields as well as their own specific fields. Furthermore, whenever you have a pointer variable whose base type is *tBaseClass*, it is legal to load this pointer with the address of any child class of *tBaseClass*; therefore, it is perfectly reasonable to load such a pointer with the address of a *tChildClassA* or *tChildClassB* variable, e.g.,

```
var
    B1: tBaseClass;
    CA: tChildClassA;
    CB: tChildClassB;
    ptr: pointer to tBaseClass;
        .
        .
        .
    lea( ebx, B1 );
    mov( ebx, ptr );
    << Use ptr >>
        .
        .
        .
    lea( eax, CA );
    mov( ebx, ptr );
    << Use ptr >>
        .
        .
        .
    lea( eax, CB );
    mov( eax, ptr );
    << Use ptr >>
```

Since *ptr* points at an object of *tBaseClass*, you may legally (from a semantic sense) access the *i*, *j*, and *r* fields of the object where *ptr* is pointing. It is not legal to access the *c*, *b*, *w*, or *d* fields of the *tChildClassA* or *tChildClassB* objects since at any one given moment the program may not know exactly what object type *ptr* references.

In order for inheritance to work properly, the *i*, *j*, and *r* fields must appear at the same offsets all child classes as they do in *tBaseClass*. This way, an instruction of the form “mov((type tBaseClass [ebx]).i, eax);” will correct access the i field even if EBX points at an object of type *tChildClassA* or *tChildClassB*. Figure 10.6 shows the layout of the child and base classes:

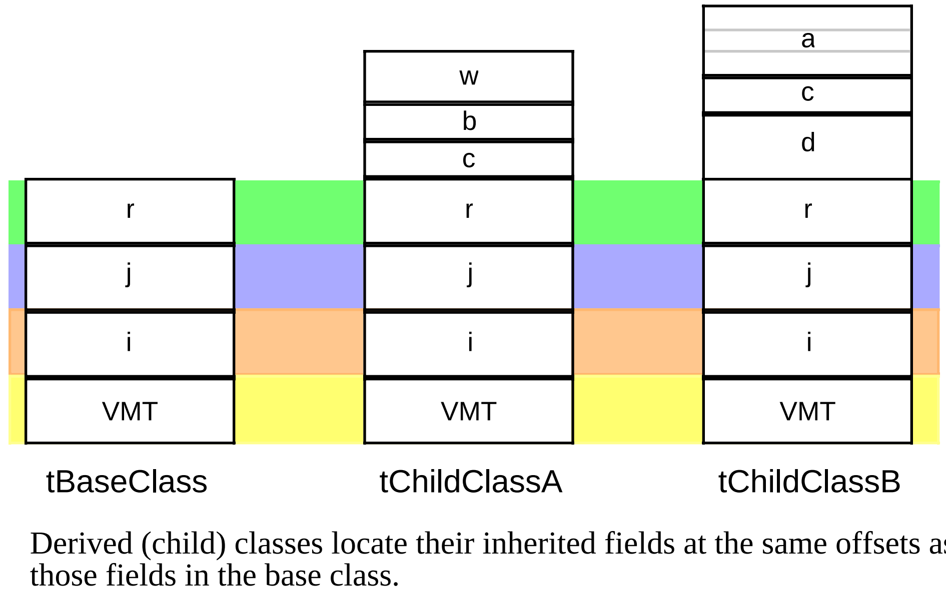

Figure 10.6 Layout of Base and Child Class Objects in Memory

Note that the new fields in the two child classes bear no relation to one another, even if they have the same name (e.g., field *c* in the two child classes does not lie at the same offset). Although the two child classes share the fields they inherit from their common base class, any new fields they add are unique and separate. Two fields in different classes share the same offset only by coincidence.

All classes (even those that aren't related to one another) place the pointer to the virtual method table at offset zero within the object. There is a single VMT associated with each class in a program; even classes that inherit fields from some base class have a VMT that is (generally) different than the base class' VMT. shows how objects of type tBaseClass, tChildClassA and tChildClassB point at their specific VMTs:


```
var
        B1: tBaseClass;
        CA: tChildClassA;
        CB: tChildClassB;
        CB2: tChildClassB;
        CA2: tChildClassA;
```

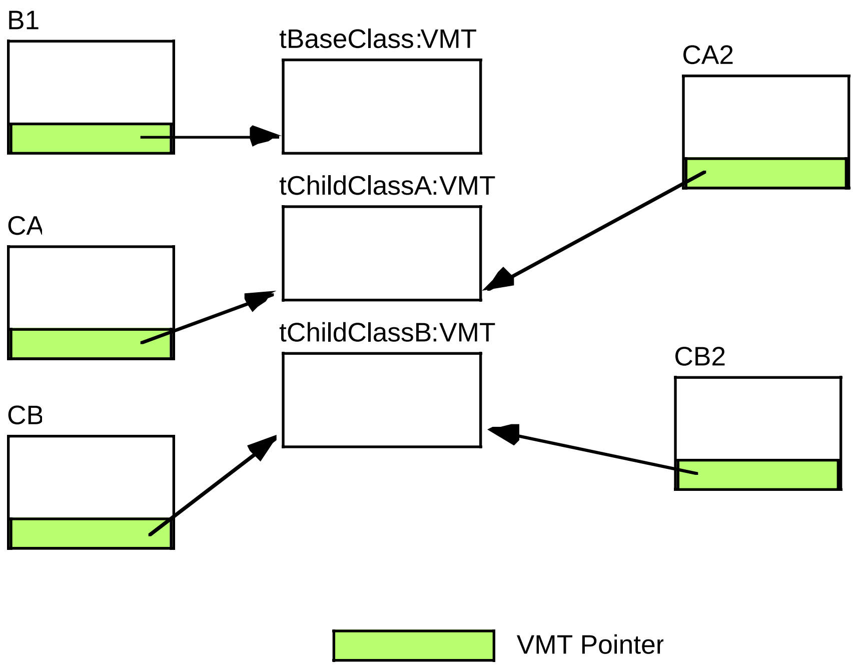

Figure 10.7 Virtual Method Table References from Objects

A virtual method table is nothing more than an array of pointers to the methods and iterators associated with a class. The address of the first method or iterator appearing in a class is at offset zero, the address of the second appears at offset four, etc. You can determine the offset value for a given iterator or method by using the @offset function. If you want to call a method or iterator directly (using 80x86 syntax rather than HLA's high level syntax), you code use code like the following:

```
var
    sc: tBaseClass;
        .
        .
        .
    lea( esi, sc );              // Get the address of the object (& VMT).
    mov( [esi], edi );           // Put address of VMT into EDI.
    call( (type dword [edi+@offset( tBaseClass.mBase )] );
```

Of course, if the method has any parameters, you must push them onto the stack before executing the code above. Don't forget, when making direct calls to a method, that you must load ESI with the address of the object. Any field references within the method will probably depend upon ESI containing this address. The choice of EDI to contain the VMT address is nearly arbitrary. Unless you're doing something tricky (like using EDI to obtain run-time type information), you could use any register you please here. As a general rule, you should use EDI when simulating class iterator/method calls because this is the convention that HLA employs and most programmers will expect this.

Whenever a child class inherits fields from some base class, the child class' VMT also inherits entries from the base class' VMT. For example, the VMT for class *tBaseClass* contains only a single entry – a pointer to method *tBaseClass.mBase*. The VMT for class *tChildClassA* contains two entries: a pointer to *tBaseClass.mBase* and *tChildClassA.mA*. Since *tChildClassB* doesn't define any new methods or iterators, *tChildClassB's* VMT contains only a single entry, a pointer to the *tBaseClass.mBase* method. Note that *tChildClassB's* VMT is identical to *tBaseClass'* VMT. Nevertheless, HLA produces two distinct VMTs. This is a critical fact that we will make use of a little later. Figure 10.8 shows the relationship between these VMTs:

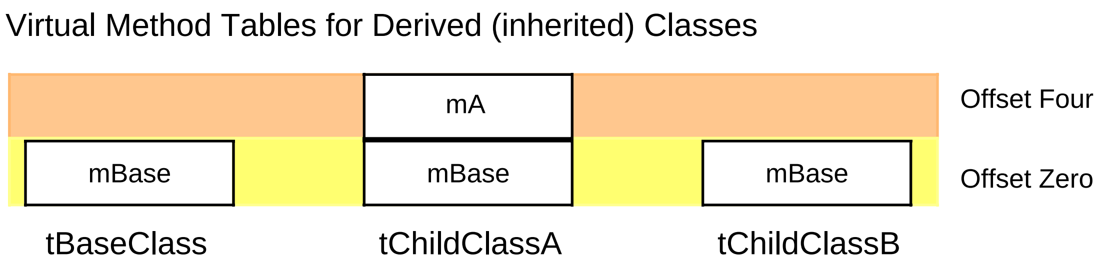

Figure 10.8 Virtual Method Tables for Inherited Classes

Although the VMT always appears at offset zero in an object (and, therefore, you can access the VMT using the address expression "[ESI]" if ESI points at an object), HLA actually inserts a symbol into the symbol table so you may refer to the VMT symbolically. The symbol *_pVMT_* (pointer to Virtual Method Table) provides this capability. So a more readable way to access the VMT pointer (as in the previous code example) is

```
lea( esi, sc );
mov( (type tBaseClass [esi])._pVMT_, edi );
call( (type dword [edi+@offset( tBaseClass.mBase )] );
```

If you need to access the VMT directly, there are a couple ways to do this. Whenever you declare a class object, HLA automatically includes a field named *_VMT_* as part of that class. *_VMT_* is a static array of double word objects. Therefore, you may refer to the VMT using an identifier of the form *classname._VMT_*. Generally, you shouldn't access the VMT directly, but as you'll see shortly, there are some good reasons why you need to know the address of this object in memory.

10.10 Constructors and Object Initialization

If you've tried to get a little ahead of the game and write a program that uses objects prior to this point, you've probably discovered that the program inexplicably crashes whenever you attempt to run it. We've covered a lot of material in this chapter thus far, but you are still missing one crucial piece of information – how to properly initialize objects prior to use. This section will put the final piece into the puzzle and allow you to begin writing programs that use classes.

Consider the following object declaration and code fragment:

```
var
    bc: tBaseClass;
        .
        .
        .
    bc.mBase();
```

Remember that variables you declare in the VAR section are uninitialized at run-time. Therefore, when the program containing these statements gets around to executing *bc.mBase*, it executes the three-statement sequence you've seen several times already:

```
lea( esi, bc);
mov( [esi], edi );
call( (type dword [edi+@offset( tBaseClass.mBase )] );
```

The problem with this sequence is that it loads EDI with an undefined value assuming you haven't previously initialized the *bc* object. Since EDI contains a garbage value, attempting to call a subroutine at address "[EDI+@offset(tBaseClass.mBase)]" will likely crash the system. Therefore, before using an object, you must initialize the *_pVMT_* field with the address of that object's VMT. One easy way to do this is with the following statement:

```
mov( &tBaseClass._VMT_, bc._pVMT_ );
```

Always remember, **before using an object, be sure to initialize the virtual method table pointer for that field**.

Although you must initialize the virtual method table pointer for all objects you use, this may not be the only field you need to initialize in those objects. Each specific class may have its own application-specific initialization that is necessary. Although the initialization may vary by class, you need to perform the same initialization on each object of a specific class that you use. If you ever create more than a single object from a given class, it is probably a good idea to create a procedure to do this initialization for you. This is such a common operation that object-oriented programmers have given these initialization procedures a special name: *constructors*.

Some object-oriented languages (e.g., C++) use a special syntax to declare a constructor. Others (e.g., Delphi) simply use existing procedure declarations to define a constructor. One advantage to employing a special syntax is that the language knows when you define a constructor and can automatically generate code to call that constructor for you (whenever you declare an object). Languages, like Delphi, require that you explicitly call the constructor; this can be a minor inconvenience and a source of defects in your programs. HLA does not use a special syntax to declare constructors – you define constructors using standard class procedures. As such, you will need to explicitly call the constructors in your program; however, you'll see an easy method for automating this in a later section of this chapter.

Perhaps the most important fact you must remember is that **constructors must be class procedures**. You must not define constructors as methods (or iterators). The reason is quite simple: one of the tasks of the constructor is to initialize the pointer to the virtual method table and you cannot call a class method or iterator until after you've initialized the VMT pointer. Since class procedures don't use the virtual method table, you can call a class procedure prior to initializing the VMT pointer for an object.

By convention, HLA programmers use the name *Create* for the class constructor. There is no requirement that you use this name, but by doing so you will make your programs easier to read and follow by other programmers.

As you may recall, you can call a class procedure via an object reference or a class reference. E.g., if *clsProc* is a class procedure of class *tClass* and *Obj* is an object of type *tClass*, then the following two class procedure invocations are both legal:

```
tClass.clsProc();
Obj.clsProc();
```

There is a big difference between these two calls. The first one calls *clsProc* with ESI containing zero (NULL) while the second invocation loads the address of *Obj* into ESI before the call. We can use this fact to determine within a method the particular calling mechanism.

10.10.1 Dynamic Object Allocation Within the Constructor

As it turns out, most programs allocate objects dynamically using *malloc* and refer to those objects indirectly using pointers. This adds one more step to the initialization process – allocating storage for the object. The constructor is the perfect place to allocate this storage. Since you probably won't need to allocate all objects dynamically, you'll need two types of constructors: one that allocates storage and then initializes the object, and another that simply initializes an object that already has storage.

Another constructor convention is to merge these two constructors into a single constructor and differentiate the type of constructor call by the value in ESI. On entry into the class' *Create* procedure, the program checks the value in ESI to see if it contains NULL (zero). If so, the constructor calls *malloc* to allocate storage for the object and returns a pointer to the object in ESI. If ESI does not contain NULL upon entry into the procedure, then the constructor assumes that ESI points at a valid object and skips over the memory allocation statements. At the very least, a constructor initializes the pointer to the VMT; therefore, the minimalist constructor will look like the following:

```
procedure tBaseClass.mBase; nodisplay;
begin mBase;

    if( ESI = 0 ) then

        push( eax );   // Malloc returns its result here, so save it.
        malloc( @size( tBaseClass ));
        mov( eax, esi );  // Put pointer into ESI;
        pop( eax );

    endif;

    // Initialize the pointer to the VMT:
    // (remember, "this" is shorthand for (type tBaseClass [esi])"

    mov( &tBaseClass._VMT_, this._pVMT_ );

    // Other class initialization would go here.

end mBase;
```

After you write a constructor like the one above, you choose an appropriate calling mechanism based on whether your object's storage is already allocated. For pre-allocated objects (i.e., those you've declared in VAR, STATIC, or STORAGE sections[6] or those you've previously allocated storage for via *malloc*) you simply load the address of the object into ESI and call the constructor. For those objects you declare as a variable, this is very easy – just call the appropriate *Create* constructor:

```
var
    bc0: tBaseClass;
    bcp: pointer to tBaseClass;
        .
        .
        .
    bc0.Create();  // Initializes pre-allocated bc0 object.
        .
        .
        .
    malloc( @size( tBaseClass ));  // Allocate storage for bcp object.
    mov( eax, bcp );
        .
        .
```

6. You generally do not declare objects in READONLY sections because you cannot initialize them.

```
        .
    bcp.Create();  // Initializes pre-allocated bcp object.
```

Note that although *bcp* is a pointer to a *tBaseClass* object, the *Create* method does not automatically allocate storage for this object. The program already allocates the storage earlier. Therefore, when the program calls *bcp.Create* it loads ESI with the address contained within *bcp*; since this is not NULL, the *tBaseClass.Create* procedure does not allocate storage for a new object. By the way, the call to *bcp.Create* emits the following sequence of machine instructions:

```
    mov( bcp, esi );
    call tBaseClass.Create;
```

Until now, the code examples for a class procedure call always began with an LEA instruction. This is because all the examples to this point have used object variables rather than pointers to object variables. Remember, a class procedure (method/iterator) call passes the address of the object in the ESI register. For object variables HLA emits an LEA instruction to obtain this address. For pointers to objects, however, the actual object address is the *value* of the pointer variable; therefore, to load the address of the object into ESI, HLA emits a MOV instruction that copies the value of the pointer into the ESI register.

In the example above, the program preallocates the storage for an object prior to calling the object constructor. While there are several reasons for preallocating object storage (e.g., you're creating a dynamic array of objects), you can achieve most simple object allocations like the one above by calling a standard *Create* method (i.e., one that allocates storage for an object if ESI contains NULL). The following example demonstrates this:

```
var
    bcp2: pointer to tBaseClass;
        .
        .
        .
    tBaseClass.Create();   // Calls Create with ESI=NULL.
    mov( esi, bcp2 );      // Save pointer to new class object in bcp2.
```

Remember, a call to a *tBaseClass.Create* constructor returns a pointer to the new object in the ESI register. It is the caller's responsibility to save the pointer this function returns into the appropriate pointer variable; the constructor does not automatically do this for you.

10.10.2 Constructors and Inheritance

Constructors for derived (child) classes that inherit fields from a base class represent a special case. Each class must have its own constructor but needs the ability to call the base class constructor. This section explains the reasons for this and how to do this.

A derived class inherits the *Create* procedure from its base class. However, you must override this procedure in a derived class because the derived class probably requires more storage than the base class and, therefore, you will probably need to use a different call to *malloc* to allocate storage for a dynamic object. Hence, it is very unusual for a derived class not to override the definition of the *Create* procedure.

However, overriding a base class' *Create* procedure has problems of its own. When you override the base class' *Create* procedure, you take the full responsibility of initializing the (entire) object, including all the initialization required by the base class. At the very least, this involves putting duplicate code in the overridden procedure to handle the initialization usually done by the base class constructor. In addition to make your program larger (by duplicating code already present in the base class constructor), this also violates information hiding principles since the derived class must be aware of all the fields in the base class (including those that are logically private to the base class). What we need here is the ability to call a base class' constructor from within the derived class' destructor and let that call do the lower-level initialization of the base class' fields. Fortunately, this is an easy thing to do in HLA.

Consider the following class declarations (which does things the hard way):

```
type
    tBase: class
        var
            i:uns32;
            j:int32;

        procedure Create(); returns( "esi" );
    endclass;

    tDerived: class inherits( tBase );
        var
            r: real64;

        override procedure Create(); returns( "esi" );
    endclass;

    procedure tBase.Create; @nodisplay;
    begin Create;

        if( esi = 0 ) then

            push( eax );
            mov( malloc( @size( tBase )), esi );
            pop( eax );

        endif;
        mov( &tBase._VMT_, this._pVMT_ );
        mov( 0, this.i );
        mov( -1, this.j );

    end Create;

    procedure tDerived.Create; @nodisplay;
    begin Create;

        if( esi = 0 ) then

            push( eax );
            mov( malloc( @size( tDerived )), esi );
            pop( eax );

        endif;

        // Initialize the VMT pointer for this object:

        mov( &tDerived._VMT_, this._pVMT_ );

        // Initialize the "r" field of this particular object:

        fldz();
        fstp( this.r );

        // Duplicate the initialization required by tBase.Create:

        mov( 0, this.i );
        mov( -1, this.j );

    end Create;
```

Let's take a closer look at the *tDerived.Create* procedure above. Like a conventional constructor, it begins by checking ESI and allocates storage for a new object if ESI contains NULL. Note that the size of a *tDerived* object includes the size required by the inherited fields, so this properly allocates the necessary storage for all fields in a *tDerived* object.

Next, the *tDerived.Create* procedure initializes the VMT pointer field of the object. Remember, each class has its own VMT and, specifically, derived classes do not use the VMT of their base class. Therefore, this constructor must initialize the _pVMT_ field with the address of the *tDerived* VMT.

After initializing the VMT pointer, the *tDerived* constructor initializes the value of the *r* field to 0.0 (remember, FLDZ loads zero onto the FPU stack). This concludes the *tDerived*-specific initialization.

The remaining instructions in *tDerived.Create* are the problem. These statements duplicate some of the code appearing in the *tBase.Create* procedure. The problem with code duplication becomes really apparent when you decide to modify the initial values of these fields; if you've duplicated the initialization code in derived classes, you will need to change the initialization code in more than one *Create* procedure. More often than not, this results in defects in the derived class *Create* procedures, especially if those derived classes appear in different source files than the base class.

Another problem with burying base class initialization in derived class constructors is the violation of the information hiding principle. Some fields of the base class may be *logically private*. Although HLA does not explicitly support the concept of public and private fields in a class (as, say, C++ does), well-disciplined programmers will still partition the fields as private or public and then only use the private fields in class routines belonging to that class. Initializing these private fields in derived classes is not acceptable to such programmers. Doing so will make it very difficult to change the definition and implementation of some base class at a later date.

Fortunately, HLA provides an easy mechanism for calling the inherited constructor within a derived class' constructor. All you have to do is call the base constructor using the classname syntax, e.g., you could call *tBase.Create* directly from within *tDerived.Create*. By calling the base class constructor, your derived class constructors can initialize the base class fields without worrying about the exact implementation (or initial values) of the base class.

Unfortunately, there are two types of initialization that every (conventional) constructor does that will affect the way you call a base class constructor: all conventional constructors allocate memory for the class if ESI contains zero and all conventional constructors initialize the VMT pointer. Fortunately, it is very easy to deal with these two problems

The memory required by an object of some most base class is usually less than the memory required for an object of a class you derive from that base class (because the derived classes usually add more fields). Therefore, you cannot allow the base class constructor to allocate the storage when you call it from inside the derived class' constructor. This problem is easily solved by checking ESI within the derived class constructor and allocating any necessary storage for the object *before* calling the base class constructor.

The second problem is the initialization of the VMT pointer. When you call the base class' constructor, it will initialize the VMT pointer with the address of the base class' virtual method table. A derived class object's *_pVMT_* field, however, must point at the virtual method table for the derived class. Calling the base class constructor will always initialize the *_pVMT_* field with the wrong pointer; to properly initialize the *_pVMT_* field with the appropriate value, the derived class constructor must store the address of the derived class' virtual method table into the *_pVMT_* field after the call to the base class constructor (so that it overwrites the value written by the base class constructor).

The *tDerived.Create* constructor, rewritten to call the *tBase.Create* constructors, follows:

```
procedure tDerived.Create; @nodisplay;
begin Create;

    if( esi = 0 ) then

        push( eax );
        mov( malloc( @size( tDerived )), esi );
        pop( eax );

    endif;
```

```
    // Call the base class constructor to do any initialization
    // needed by the base class.  Note that this call must follow
    // the object allocation code above (so ESI will always contain
    // a pointer to an object at this point and tBase.Create will
    // never allocate storage).

    tBase.Create();

    // Initialize the VMT pointer for this object.  This code
    // must always follow the call to the base class constructor
    // because the base class constructor also initializes this
    // field and we don't want the initial value supplied by
    // tBase.Create.

    mov( &tDerived._VMT_, this._pVMT_ );

    // Initialize the "r" field of this particular object:

    fldz();
    fstp( this.r );

end Create;
```

This solution solves all the above concerns with derived class constructors.

10.10.3 Constructor Parameters and Procedure Overloading

All the constructor examples to this point have not had any parameters. However, there is nothing special about constructors that prevent the use of parameters. Constructors are procedures therefore you can specify any number and types of parameters you choose. You can use these parameter values to initialize certain fields or control how the constructor initializes the fields. Of course, you may use constructor parameters for any purpose you'd use parameters in any other procedure. In fact, about the only issue you need concern yourself with is the use of parameters whenever you have a derived class. This section deals with those issues.

The first, and probably most important, problem with parameters in derived class constructors actually applies to all overridden procedures, iterators, and methods: the parameter list of an overridden routine must exactly match the parameter list of the corresponding routine in the base class. In fact, HLA doesn't even give you the chance to violate this rule because OVERRIDE routine prototypes don't allow parameter list declarations – they automatically inherit the parameter list of the base routine. Therefore, you cannot use a special parameter list in the constructor prototype for one class and a different parameter list for the constructors appearing in base or derived classes. Sometimes it would be nice if this weren't the case, but there are some sound and logical reasons why HLA does not support this[7].

Some languages, like C++, support function overloading letting you specify several different constructors whose parameter list specifies which constructor to use. HLA does not directly support procedure overloading in this manner, but you can use macros to simulate this language feature (see "Simulating Function Overloading with Macros" on page 990). To use this trick with constructors you would create a macro with the name *Create*. The actual constructors could have names that describe their differences (e.g., *CreateDefault*, *CreateSetIJ*, etc.). The *Create* macro would parse the actual parameter list to determine which routine to call.

7. Calling virtual methods and iterators would be a real problem since you don't really know which routine a pointer references. Therefore, you couldn't know the proper parameter list. While the problems with procedures aren't quite as drastic, there are some subtle problems that could creep into your code if base or derived classes allowed overridden procedures with different parameter lists.

HLA does not support macro overloading. Therefore, you cannot override a macro in a derived class to call a constructor unique to that derived class. In certain circumstances you can create a small workaround by defining empty procedures in your base class that you intend to override in some derived class (this is similar to an abstract method, see “Abstract Methods” on page 1091). Presumably, you would never call the procedure in the base class (in fact, you would probably want to put an error message in the body of the procedure just in case you accidentally call it). By putting the empty procedure declaration in the base class, the macro that simulates function overloading can refer to that procedure and you can use that in derived classes later on.

10.11 Destructors

A destructor is a class routine that cleans up an object once a program finishes using that object. Like constructors, HLA does not provide a special syntax for creating destructors nor does HLA automatically call a destructor; unlike constructors, a destructor is usually a method rather than a procedure (since virtual destructors make a lot of sense while virtual constructors do not).

A typical destructor will close any files opened by the object, free the memory allocated during the use of the object, and, finally, free the object itself if it was created dynamically. The destructor also handles any other clean-up chores the object may require before it ceases to exist.

By convention, most HLA programmers name their destructors *Destroy.* Destructors generally do not have any parameters, so the issue of overloading the parameter list rarely arises. About the only code that most destructors have in common is the code to free the storage associated with the object. The following destructor demonstrates how to do this:

```
procedure tBase.Destroy; nodisplay;
begin Destroy;

    push( eax );   // isInHeap uses this

    // Place any other clean up code here.
    // The code to free dynamic objects should always appear last
    // in the destructor.

        /*************/

    // The following code assumes that ESI still contains the address
    // of the object.

    if( isInHeap( esi )) then

        free( esi );

    endif;
    pop( eax );

end Destroy;
```

The HLA Standard Library routine *isInHeap* returns true if its parameter is an address that *malloc* returned. Therefore, this code automatically frees the storage associated with the object if the program originally allocated storage for the object by calling *malloc*. Obviously, on return from this method call, ESI will no longer point at a legal object in memory if you allocated it dynamically. Note that this code will not affect the value in ESI nor will it modify the object if the object wasn’t one you’ve previously allocated via a call to *malloc*.

10.12 HLA's "_initialize_" and "_finalize_" Strings

Although HLA does not automatically call constructors and destructors associated with your classes, HLA does provide a mechanism whereby you can cause these calls to happen automatically: by using the *_initialize_* and *_finalize_* compile-time string variables (i.e., VAL constants) HLA automatically declares in every procedure.

Whenever you write a procedure, iterator, or method, HLA automatically declares several local symbols in that routine. Two such symbols are *_initialize_* and *_finalize_*. HLA declares these symbols as follows:

```
val
    _initialize_: string := "";
    _finalize_: string := "";
```

HLA emits the *_initialize_* string as text at the very beginning of the routine's body, i.e., immediately after the routine's BEGIN clause[8]. Similarly, HLA emits the *_finalize_* string at the very end of the routine's body, just before the END clause. This is comparable to the following:

```
procedure SomeProc;
    << declarations >>
begin SomeProc;

    @text( _initialize_ );

        << procedure body >>

    @text( _finalize_ );

end SomeProc;
```

Since *_initialize_* and *_finalize_* initially contain the empty string, these expansions have no effect on the code that HLA generates unless you explicitly modify the value of *_initialize_* prior to the BEGIN clause or you modify *_finalize_* prior to the END clause of the procedure. So if you modify either of these string objects to contain a machine instruction, HLA will compile that instruction at the beginning or end of the procedure. The following example demonstrates how to use this technique:

```
procedure SomeProc;
    ?_initialize_ := "mov( 0, eax );";
    ?_finalize_ := "stdout.put( eax );"
begin SomeProc;

    // HLA emits "mov( 0, eax );" here in response to the _initialize_
    // string constant.

    add( 5, eax );

    // HLA emits "stdout.put( eax );" here.

end SomeProc;
```

Of course, these examples don't save you much. It would be easier to type the actual statements at the beginning and end of the procedure than assign a string containing these statements to the *_initialize_* and *_finalize_* compile-time variables. However, if we could automate the assignment of some string to these variables, so that you don't have to explicitly assign them in each procedure, then this feature might be useful. In a moment, you'll see how we can automate the assignment of values to the *_initialize_* and *_finalize_* strings. For the time being, consider the case where we load the name of a constructor into the *_initialize_*

8. If the routine automatically emits code to construct the activation record, HLA emits *_initialize_*'s text after the code that builds the activation record.

string and we load the name of a destructor in to the *_finalize_* string. By doing this, the routine will "automatically" call the constructor and destructor for that particular object.

The example above has a minor problem. If we can automate the assignment of some value to *_initialize_* or *_finalize_*, what happens if these variables already contain some value? For example, suppose we have two objects we use in a routine and the first one loads the name of its constructor into the *_initialize_* string; what happens when the second object attempts to do the same thing? The solution is simple: don't directly assign any string to the *_initialize_* or *_finalize_* compile-time variables, instead, always concatenate your strings to the end of the existing string in these variables. The following is a modification to the above example that demonstrates how to do this:

```
procedure SomeProc;
    ?_initialize_ := _initialize_  + "mov( 0, eax );";
    ?_finalize_ := _finalize_ + "stdout.put( eax );"
begin SomeProc;

    // HLA emits "mov( 0, eax );" here in response to the _initialize_
    // string constant.

    add( 5, eax );

    // HLA emits "stdout.put( eax );" here.

end SomeProc;
```

When you assign values to the *_initialize_* and *_finalize_* strings, HLA almost guarantees that the *_initialize_* sequence will execute upon entry into the routine. Sadly, the same is not true for the *_finalize_* string upon exit. HLA simply emits the code for the *_finalize_* string at the end of the routine, immediately before the code that cleans up the activation record and returns. Unfortunately, "falling off the end of the routine" is not the only way that one could return from that routine. One could explicitly return from somewhere in the middle of the code by executing a RET instruction. Since HLA only emits the *_finalize_* string at the very end of the routine, returning from that routine in this manner bypassing the *_finalize_* code. Unfortunately, other than manually emitting the *_finalize_* code, there is nothing you can do about this[9]. Fortunately, this mechanism for exiting a routine is completely under your control; if you never exit a routine except by "falling off the end" then you won't have to worry about this problem (note that you can use the EXIT control structure to transfer control to the end of a routine if you really want to return from that routine from somewhere in the middle of the code).

Another way to prematurely exit a routine which, unfortunately, you have no control over, is by raising an exception. Your routine could call some other routine (e.g., a standard library routine) that raises an exception and then transfers control immediately to whomever called your routine. Fortunately, you can easily trap and handle exceptions by putting a TRY..ENDTRY block in your procedure. Here is an example that demonstrates this:

```
procedure SomeProc;
    << declarations that  modify _initialize_ and _finalize_ >>
begin SomeProc;

    << HLA emits the code for the _initialize_ string here. >>

    try   // Catch any exceptions that occur:

        << Procedure Body Goes Here >>

      anyexception

        push( eax );   // Save the exception #.
        @text( _finalize_ );  // Execute the _finalize_ code here.
```

9. Note that you can manually emit the *_finalize_* code using the statement "@text(_finalize_);".

```
        pop( eax );             // Restore the exception #.
        raise( eax );           // Reraise the exception.

    endtry;

    // HLA automatically emits the _finalize_ code here.

end SomeProc;
```

Although the code above handles some problems that exist with *_finalize_*, by no means that this handle every possible case. Always be on the look out for ways your program could inadvertently exit a routine without executing the code found in the *_finalize_* string. You should explicitly expand *_finalize_* if you encounter such a situation.

There is one important place you can get into trouble with respect to exceptions: within the code the routine emits for the *_initialize_* string. If you modify the _initialize_ string so that it contains a constructor call and the execution of that constructor raises an exception, this will probably force an exit from that routine without executing the corresponding *_finalize_* code. You could bury the TRY..ENDTRY statement directly into the *_initialize_* and *_finalize_* strings but this approach has several problems, not the least of which is the fact that one of the first constructors you call might raise an exception that transfers control to the exception handler that calls the destructors for all objects in that routine (including those objects whose constructors you have yet to call). Although no single solution that handles all problems exists, probably the best approach is to put a TRY..ENDTRY block around each constructor call if it is possible for that constructor to raise some exception that is possible to handle (i.e., doesn't require the immediate termination of the program).

Thus far this discussion of *_initialize_* and *_finalize_* has failed to address one important point: why use this feature to implement the "automatic" calling of constructors and destructors since it apparently involves more work that simply calling the constructors and destructors directly? Clearly there must be a way to automate the assignment of the *_initialize_* and *_finalize_* strings or this section wouldn't exist. The way to accomplish this is by using a macro to define the class type. So now it's time to take a look at another HLA feature that makes is possible to automate this activity: the FORWARD keyword.

You've seen how to use the FORWARD reserved word to create procedure and iterator prototypes (see "Forward Procedures" on page 567), it turns out that you can declare forward CONST, VAL, TYPE, and variable declarations as well. The syntax for such declarations takes the following form:

```
ForwardSymbolName: forward( undefinedID );
```

This declaration is completely equivalent to the following:

```
?undefinedID: text := "ForwardSymbolName";
```

Especially note that this expansion does not actually define the symbol *ForwardSymbolName*. It just converts this symbol to a string and assigns this string to the specified TEXT object (*undefinedID* in this example).

Now you're probably wonder how something like the above is equivalent to a forward declaration. The truth is, it isn't. However, FORWARD declarations let you create macros that simulate type names by allowing you to defer the actual declaration of an object's type until some later point in the code. Consider the following example:

```
type
    myClass: class
        var
            i:int32;

        procedure Create; returns( "esi" );
        procedure Destroy;
    endclass;

#macro _myClass: varID;
```

```
    forward( varID );
    ?_initialize_ := _initialize_ + @string:varID + ".Create(); ";
    ?_finalize_ := _finalize_ + @string:varID + ".Destroy(); ";
    varID: myClass
#endmacro;
```

Note, and this is very important, that a semicolon does not follow the "varID: myClass" declaration at the end of this macro. You'll find out why this semicolon is missing in a little bit.

If you have the class and macro declarations above in your program, you can now declare variables of type *_myClass* that automatically invoke the constructor and destructor upon entry and exit of the routine containing the variable declarations. To see how, take a look at the following procedure shell:

```
procedure HasmyClassObject;
var
    mco: _myClass;
begin HasmyClassObject;

    << do stuff with mco here >>

end HasmyClassObject;
```

Since *_myClass* is a macro, the procedure above expands to the following text during compilation:

```
procedure HasmyClassObject;
var
    mco:                    // Expansion of the _myClass macro:
        forward( _0103_ );  // _0103_ symbol is and HLA supplied text symbol
                            // that expands to "mco".

    ?_initialize_ := _initialize_ + "mco" + ".Create(); ";
    ?_finalize_ := _finalize_ + "mco" + ".Destroy(); ";
    mco: myClass;

begin HasmyClassObject;

    mco.Create();  // Expansion of the _initialize_ string.

    << do stuff with mco here >>

    mco.Destroy(); // Expansion of the _finalize_ string.

end HasmyClassObject;
```

You might notice that a semicolon appears after "mco: myClass" declaration in the example above. This semicolon is not actually a part of the macro, instead it is the semicolon that follows the "mco: _myClass;" declaration in the original code.

If you want to create an array of objects, you could legally declare that array as follows:

```
var
    mcoArray: _myClass[10];
```

Because the last statement in the _myClass macro doesn't end with a semicolon, the declaration above will expand to something like the following (almost correct) code:

```
    mcoArray:               // Expansion of the _myClass macro:
        forward( _0103_ );  // _0103_ symbol is and HLA supplied text symbol
                            // that expands to "mcoArray".

    ?_initialize_ := _initialize_ + "mcoArray" + ".Create(); ";
    ?_finalize_ := _finalize_ + "mcoArray" + ".Destroy(); ";
    mcoArray: myClass[10];
```


The only problem with this expansion is that it only calls the constructor for the first object of the array. There are several ways to solve this problem; one is to append a macro name to the end of *_initialize_* and *_finalize_* rather than the constructor name. That macro would check the object's name (*mcoArray* in this example) to determine if it is an array. If so, that macro could expand to a loop that calls the constructor for each element of the array (the implementation appears as a programming project at the end of this chapter).

Another solution to this problem is to use a macro parameter to specify the dimensions for arrays of *myClass*. This scheme is easier to implement than the one above, but it does have the drawback of requiring a different syntax for declaring object arrays (you have to use parentheses around the array dimension rather than square brackets).

The FORWARD directive is quite powerful and lets you achieve all kinds of tricks. However, there are a few problems of which you should be aware. First, since HLA emits the *_initialize_* and *_finalize_* code transparently, you can be easily confused if there are any errors in the code appearing within these strings. If you start getting error messages associated with the BEGIN or END statements in a routine, you might want to take a look at the *_initialize_* and *_finalize_* strings within that routine. The best defense here is to always append very simple statements to these strings so that you reduce the likelihood of an error.

Fundamentally, HLA doesn't support automatic constructor and destructor calls. This section has presented several tricks to attempt to automate the calls to these routines. However, the automation isn't perfect and, indeed, the aforementioned problems with the *_finalize_* strings limit the applicability of this approach. The mechanism this section presents is probably fine for simple classes and simple programs. However, one piece of advice is probably worth following: if your code is complex or correctness is critical, it's probably a good idea to explicitly call the constructors and destructors manually.

10.13 Abstract Methods

An *abstract base class* is one that exists solely to supply a set of common fields to its derived classes. You never declare variables whose type is an abstract base class, you always use one of the derived classes. The purpose of an abstract base class is to provide a template for creating other classes, nothing more. As it turns out, the only difference in syntax between a standard base class and an abstract base class is the presence of at least one *abstract method* declaration. An abstract method is a special method that does not have an actual implementation in the abstract base class. Any attempt to call that method will raise an exception. If you're wondering what possible good an abstract method could be, well, keep on reading...

Suppose you want to create a set of classes to hold numeric values. One class could represent unsigned integers, another class could represent signed integers, a third could implement BCD values, and a fourth could support *real64* values. While you could create four separate classes that function independently of one another, doing so passes up an opportunity to make this set of classes more convenient to use. To understand why, consider the following possible class declarations:

```
type
    uint: class
        var
            TheValue: dword;

        method put;
        << other methods for this class >>
    endclass;

    sint: class
        var
            TheValue: dword;

        method put;
        << other methods for this class >>
    endclass;
```

```
r64: class
    var
        TheValue: real64;

    method put;
    << other methods for this class >>
endclass;
```

The implementation of these classes is not unreasonable. They have fields for the data, they have a *put* method (which, presumably, writes the data to the standard output device), Presumably they have other methods and procedures in implement various operations on the data. There is, however, two problems with these classes, one minor and one major, both occurring because these classes do not inherit any fields from a common base class.

The first problem, which is relatively minor, is that you have to repeat the declaration of several common fields in these classes. For example, the *put* method declaration appears in each of these classes[10]. This duplication of effort involves results in a harder to maintain program because it doesn't encourage you to use a common name for a common function since it's easy to use a different name in each of the classes.

A bigger problem with this approach is that it is not generic. That is, you can't create a generic pointer to a "numeric" object and perform operations like addition, subtraction, and output on that value (regardless of the underlying numeric representation).

We can easily solve these two problems by turning the previous class declarations into a set of derived classes. The following code demonstrates an easy way to do this:

```
type
    numeric: class
        procedure put;
        << Other common methods shared by all the classes >>
    endclass;

    uint: class inherits( numeric )
        var
            TheValue: dword;

        override method put;
        << other methods for this class >>
    endclass;

    sint: class inherits( numeric )
        var
            TheValue: dword;

        override method put;
        << other methods for this class >>
    endclass;

    r64: class inherits( numeric )
        var
            TheValue: real64;

        override method put;
        << other methods for this class >>
    endclass;
```

This scheme solves both the problems. First, by inheriting the *put* method from numeric, this code encourages the derived classes to always use the name *put* thereby making the program easier to maintain. Second, because this example uses derived classes, it's possible to create a pointer to the *numeric* type and

10. Note, by the way, that *TheValue* is not a common class because this field has a different type in the *r64* class.

load this pointer with the address of a *uint*, *sint*, or *r64* object. That pointer can invoke the methods found in the *numeric* class to do functions like addition, subtraction, or numeric output. Therefore, the application that uses this pointer doesn't need to know the exact data type, it only deals with numeric values in a generic fashion.

One problem with this scheme is that it's possible to declare and use variables of type *numeric*. Unfortunately, such numeric variables don't have the ability to represent any type of number (notice that the data storage for the numeric fields actually appears in the derived classes). Worse, because you've declared the *put* method in the *numeric* class, you've actually got to write some code to implement that method even though one should never really call it; the actual implementation should only occur in the derived classes. While you could write a dummy method that prints an error message (or, better yet, raises an exception), there shouldn't be any need to write "dummy" procedures like this. Fortunately, there *is* no reason to do so – if you use *abstract* methods.

The ABSTRACT keyword, when it follows a method declaration, tells HLA that you are not going to provide an implementation of the method for this class. Instead, it is the responsibility of all derived class to provide a concrete implementation for the abstract method. HLA will raise an exception if you attempt to call an abstract method directly. The following is the modification to the *numeric* class to convert *put* to an abstract method:

```
type
    numeric: class
        method put; abstract;
        << Other common methods shared by all the classes >>
    endclass;
```

An abstract base class is a class that has at least one abstract method. Note that you don't have to make all methods abstract in an abstract base class; it is perfectly legal to declare some standard methods (and, of course, provide their implementation) within the abstract base class.

Abstract method declarations provide a mechanism by which a base class enforces the methods that the derived classes must implement. In theory, all derived classes must provide concrete implementations of all abstract methods or those derived classes are themselves abstract base classes. In practice, it's possible to bend the rules a little and use abstract methods for a slightly different purpose.

A little earlier, you read that one should never create variables whose type is an abstract base class. For if you attempt to execute an abstract method the program would immediately raise an exception to complain about this illegal method call. In practice, you actually can declare variables of an abstract base type and get away with this as long as you don't call any abstract methods. We can use this fact to provide a better form of method overloading (that is, providing several different routines with the same name but different parameter lists). Remember, the standard trick in HLA to overload a routine is to write several different routines and then use a macro to parse the parameter list and determine which actual routine to call (see "Simulating Function Overloading with Macros" on page 990). The problem with this technique is that you cannot override a macro definition in a class, so if you want to use a macro to override a routine's syntax, then that macro must appear in the base class. Unfortunately, you may not need a routine with a specific parameter list in the base class (for that matter, you may only need that particular version of the routine in a single derived class), so implementing that routine in the base class and in all the other derived classes is a waste of effort. This isn't a big problem. Just go ahead and define the abstract method in the base class and only implement it in the derived class that needs that particular method. As long as you don't call that method in the base class or in the other derived classes that don't override the method, everything will work fine.

One problem with using abstract methods to support overloading is that this trick does not apply to procedures - only methods and iterators. However, you can achieve the same effect with procedures by declaring a (non-abstract) procedure in the base class and overriding that procedure only in the class that actually uses it. You will have to provide an implementation of the procedure in the base class, but that is a minor issue (the procedure's body, by the way, should simply raise an exception to indicate that you should have never called it).

An example of routine overloading in a class appears in this chapter's sample program.

10.14 Run-time Type Information (RTTI)

When working with an object variable (as opposed to a pointer to an object), the type of that object is obvious: it's the variable's declared type. Therefore, at both compile-time and run-time the program trivially knows the type of the object. When working with pointers to objects you cannot, in the general case, determine the type of an object a pointer references. However, at run-time it is possible to determine the object's actual type. This section discusses how to detect the underlying object's type and how to use this information.

If you have a pointer to an object and that pointer's type is some base class, at run-time the pointer could point at an object of the base class or any derived type. At compile-time it is not possible to determine the exact type of an object at any instant. To see why, consider the following short example:

```
ReturnSomeObject();           // Returns a pointer to some class in ESI.
mov( esi, ptrToObject );
```

The routine *ReturnSomeObject* returns a pointer to an object in ESI. This could be the address of some base class object or a derived class object. At compile-time there is no way for the program to know what type of object this function returns. For example, *ReturnSomeObject* could ask the user what value to return so the exact type could not be determined until the program actually runs and the user makes a selection.

In a perfectly designed program, there probably is no need to know a generic object's actual type. After all, the whole purpose of object-oriented programming and inheritance is to produce general programs that work with lots of different objects without having to make substantial changes to the program. In the real world, however, programs may not have a perfect design and sometimes it's nice to know the exact object type a pointer references. Run-time type information, or RTTI, gives you the capability of determining an object's type at run-time, even if you are referencing that object using a pointer to some base class of that object.

Perhaps the most fundamental RTTI operation you need is the ability to ask if a pointer contains the address of some specific object type. Many object-oriented languages (e.g., Delphi) provide an *IS* operator that provides this functionality. IS is a boolean operator that returns true if its left operand (a pointer) points at an object whose type matches the left operand (which must be a type identifier). The typical syntax is generally the following:

```
ObjectPointerOrVar  is ClassType
```

This operator would return true if the variable is of the specified class, it returns false otherwise. Here is a typical use of this operator (in the Delphi language)

```
if( ptrToNumeric is uint ) then begin
    .
    .
    .
end;
```

It's actually quite simple to implement this functionality in HLA. As you may recall, each class is given its own virtual method table. Whenever you create an object, you must initialize the pointer to the VMT with the address of that class' VMT. Therefore, the VMT pointer field of all objects of a given class type contain the same pointer value and this pointer value is different from the VMT pointer field of all other classes. We can use this fact to see if an object is some specific type. The following code demonstrates how to implement the Delphi statement above in HLA:

```
mov( ptrToNumeric, esi );
if( (type uint [esi])._pVMT_ = &uint._VMT_  ) then
    .
    .
```



```
    .
endif;
```

This IF statement simply compares the object's *_pVMT_* field (the pointer to the VMT) against the address of the desired class' VMT. If they are equal, then the *ptrToNumeric* variable points at an object of type *uint*.

Within the body of a class method or iterator, there is a slightly easier way to see if the object is a certain class. Remember, upon entry into a method or an iterator, the EDI register contains the address of the virtual method table. Therefore, assuming you haven't modified EDI's value, you can easily test to see if THIS (ESI) is a specific class type using an IF statement like the following:

```
if( EDI = &uint._VMT_  ) then
    .
    .
    .
endif;
```

10.15 Calling Base Class Methods

In the section on constructors you saw that it is possible to call an ancestor class' procedure within the derived class' overridden procedure. To do this, all you needed to do was to invoke the procedure using the call "classname.procedureName(parameters);" On occasion you may want to do this same operation with a class' methods as well as its procedures (that is, have an overridden method call the corresponding base class method in order to do some computation you'd rather not repeat in the derived class' method). Unfortunately, HLA does not let you directly call methods as it does procedures. You will need to use an indirect mechanism to achieve this; specifically, you will have to call the function using the address in the base class' virtual method table. This section describes how to do this.

Whenever your program calls a method it does so indirectly, using the address found in the virtual method table for the method's class. The virtual method table is nothing more than an array of 32-bit pointers with each entry containing the address of one of that class' methods. So to call a method, all you need is the index into this array (or, more properly, the offset into the array) of the address of the method you wish to call. The HLA compile-time function *@offset* comes to the rescue- it will return the offset into the virtual method table of the method whose name you supply as a parameter. Combined with the CALL instruction, you can easily call any method associated with a class. Here's an example of how you would do this:

```
type
    myCls: class
               .
               .
               .
               method m;
               .
               .
               .
    endclass;
        .
        .
        .
    call( myCls._VMT_[ @offset( myCls.m )]);
```

The CALL instruction above calls the method whose address appears at the specified entry in the virtual method table for *myCls*. The *@offset* function call returns the offset (i.e., index times four) of the address of *myCls.m* within the virtual method table. Hence, this code indirectly calls the *m* method by using the virtual method table entry for *m*.

There is one major drawback to calling methods using this scheme: you don't get to use the high level syntax for procedure/method calls. Instead, you must use the low-level CALL instruction. In the example above, this isn't much of an issue because the *m* procedure doesn't have any parameters. If it did have parameters, you would have to manually push those parameters onto the stack yourself (see "Passing Parameters on the Stack" on page 822). Fortunately, you'll rarely need to call ancestor class methods from a derived class, so this won't be much of an issue in real-world programs.

10.16 Sample Program

This chapter's sample program will present what is probably the epitome of object-oriented programs: a simple "drawing" program that uses objects to represent shapes to draw on the display. While limited to a demonstration program, this program does demonstrate important object-oriented concepts in assembly language.

This is an unusual drawing program insofar as it draws shapes using ASCII characters. While the shapes it draws are very rough (compared to a graphics-based drawing program), the output of this program could be quite useful for creating rudimentary diagrams to include as comments in your HLA (or other language) programs. This sample program does not provide a "user interface" for drawing images (something you would need to effectively use this program) because the user interface represents a lot of code that won't improve your appreciation of object-oriented programming (not to mention, this book is long enough already). Providing a mouse-based user interface to this program is left as an exercise to the interested reader.

This program consists of three source files: the class definitions in a header file, the implementation of the class' procedures and methods in an HLA source file, and a main program that demonstrates a simple use of the class' objects. The following listings are for these three files.

```
type

    // Generic shape class:

    shape: class

        const

            maxX: uns16 := 80;
            maxY: uns16 := 25;

        var
            x:            uns16;
            y:            uns16;
            width:     uns16;
            height:    uns16;
            fillShape:boolean;

        procedure create; returns( "esi" ); external;

        method draw; abstract;
        method fill( f:boolean ); external;
        method moveTo( x:uns16; y:uns16 );  external;
        method resize( width: uns16; height: uns16 );  external;

    endclass;


    // Class for a rectangle shape
```

```
//
// +------+
// |      |
//  +------+

rect: class inherits( shape )

    override procedure create;  external;
    override method draw; external;

endclass;



// Class for a rounded rectangle shape
//
//   --------
// /         \
//  |         |
//  \        /
//   --------

roundrect: class inherits( shape )

    override procedure create; external;
    override method draw;  external;

endclass;


// Class for a diamond shape
//
//      /\
//     /  \
//      \  /
//       \/

diamond: class inherits( shape )

    override procedure create; external;
    override method resize; external;
    override method draw;  external;

endclass;
```

Program 10.1 Shapes.hhf - The Shape Class Header Files

```
unit Shapes;
#includeonce( "stdlib.hhf" )
#includeonce( "shapes.hhf" )


// Emit the virtual method tables for the classes:

static
```

```
    vmt( shape );
    vmt( rect );
    vmt( roundrect );
    vmt( diamond );



/*********************************************************************/

// Generic shape methods and procedures



// Constructor for the shape class.
//
//  Note: this should really be an abstract procedure, but since
//  HLA doesn't support abstract procedures we'll fake it by
//  raising an exception if somebody tries to call this proc.

procedure shape.create; @nodisplay; @noframe;
begin create;

    // This should really be an abstract procedure,
    // but such things don't exist, so we will fake it.

    raise( ex.ExecutedAbstract );

end create;


// Generic shape.fill method.
// This is an accessor function that sets the "fill" field
// to the value of the parameter.

method shape.fill( f:boolean ); @nodisplay;
begin fill;

    push( eax );
    mov( f, al );
    mov( al, this.fillShape );
    pop( eax );

end fill;

// Generic shape.moveTo method.
// Checks the coordinates passed as a parameter and
// then sets the (X,Y) coordinates of the underlying
// shape object to these values.

method shape.moveTo( x:uns16; y:uns16 ); @nodisplay;
begin moveTo;

    push( eax );
    push( ebx );


    mov( x, ax );
    assert( ax < shape.maxX );
    mov( ax, this.x );

    mov( y, ax );
```

```
    assert( ax < shape.maxY );
    mov( ax, this.y );

    pop( ebx );
    pop( eax );

end moveTo;


// Generic shape.resize method.
// Sets the width and height fields of the underlying object
// to the values passed as parameters.
//
//  Note: Ignores resize request if the size is less than 2x2.

method shape.resize( width:uns16; height:uns16 ); @nodisplay;
begin resize;

    push( eax );
    assert( width <= shape.maxX );
    assert( height <= shape.maxY );

    if( width > 2 ) then

        if( height > 2 ) then

            mov( width, ax );
            mov( ax, this.width );


            mov( height, ax );
            mov( ax, this.height );

        endif;

    endif;
    pop( eax );

end resize;


/*******************/
/*                 */
/* rect's methods: */
/*                 */
/*******************/


// Constructor for the rectangle class:

procedure rect.create; @nodisplay; @noframe;
begin create;

    push( eax );

    // If called as rect.create, then allocate a new object
    // on the heap and return the pointer in ESI.

    if( esi = NULL ) then

        mov( malloc( @size( rect ) ), esi );
```

```
        endif;

        // Initialize the pointer to the VMT:

        mov( &rect._VMT_, this._pVMT_ );

        // Initialize fields to create a non-filled unit square.

        sub( eax, eax );

        mov( ax, this.x );
        mov( ax, this.y );
        inc( eax );
        mov( al, this.fillShape );  // Sets fillShape to true.
        inc( eax );
        mov( ax, this.height );
        mov( ax, this.width );

        pop( eax );
        ret();

    end create;


    // Here's the method to draw a text-based square on the display.

    method rect.draw; @nodisplay;
    static
        horz: str.strvar( shape.maxX ); // Holds "+------...--+"
        spcs: str.strvar( shape.maxX ); // Holds "      ...    " for fills.


    begin draw;

        push( eax );
        push( ebx );
        push( ecx );
        push( edx );

        // Initialize the horz and spcs strings to speed up
        // drawing our rectangle.

        movzx( this.width, ebx );
        str.setstr( '-', horz, ebx );
        mov( horz, eax );
        mov( '+', (type char [eax]));
        mov( '+', (type char [eax+ebx-1]));

        // If the fillShape field contains true, then we
        // need to fill in the characters inside the rectangle.
        // If this is false, we don't want to overwrite the
        // text in the center of the rectangle.  The following
        // code initializes spcs to all spaces or the empty string
        // to accomplish this.

        if( this.fillShape ) then

            sub( 2, ebx );
            str.setstr( ' ', spcs, ebx );
```



```
        else

            str.cpy( "", spcs );

        endif;

        // Okay, position the cursor and draw
        // our rectangle.

        console.gotoxy( this.y, this.x );
        stdout.puts( horz );    // Draws top horz line.

        // For each row except the top and bottom rows,
        // draw "|" characters on the left and right
        // hand sides and the fill characters (if fillShape
        // is true) inbetween them.

        mov( this.y, cx );
        mov( cx, bx );
        add( this.height, bx );
        inc( cx );
        dec( bx );
        while( cx < bx) do

            console.gotoxy( cx, this.x );
            stdout.putc( '|' );
            stdout.puts( spcs );
            mov( this.x, dx );
            add( this.width, dx );
            dec( dx );
            console.gotoxy( cx, dx );
            stdout.putc( '|' );
            inc( cx );

        endwhile;

        // Draw the bottom horz bar:

        console.gotoxy( cx, this.x );
        stdout.puts( horz );

        pop( edx );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end draw;




    /*************************/
    /*                       */
    /* roundrect's methods: */
    /*                       */
    /*************************/
```

```
// This is the constructor for the roundrect class.
// See the comments in rect.create for details
// (since this is just a clone of that code with
//  minor changes here and there).

procedure roundrect.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = NULL ) then

        mov( malloc( @size( rect ) ), esi );

    endif;
    mov( &roundrect._VMT_, this._pVMT_ );

    // Initialize fields to create a non-filled unit square.

    sub( eax, eax );

    mov( ax, this.x );
    mov( ax, this.y );
    inc( eax );
    mov( al, this.fillShape );  // Sets fillShape to true.
    inc( eax );
    mov( ax, this.height );
    mov( ax, this.width );

    pop( eax );
    ret();

end create;



// Here is the draw method for the roundrect object.
// Note: if the object is less than 5x4 in size,
// this code calls rect.draw to draw a rectangle
// since roundrects smaller than 5x4 don't look good.
//
//  Typical roundrect:
//
//       --------
//      /        \
//      |        |
//      \        /
//       --------

method roundrect.draw; @nodisplay;
static
    horz:       str.strvar( shape.maxX );
    spcs:       str.strvar( shape.maxX );

begin draw;

    push( eax );
    push( ebx );
    push( ecx );
    push( edx );

    if
```

```
( #{
    cmp( this.width, 5 );
    jb true;
    cmp( this.height, 4 );
    jae false;
}#) then

    // If it's too small to draw an effective
    // roundrect, then draw it as a rectangle.

    call( rect._VMT_[ @offset( rect.draw ) ] );

else

    // Okay, it's big enough, draw it as a rounded
    // rectangle object.  Begin by initializing the
    // horz string with a set of dashes with spaces
    // at either end.

    movzx( this.width, ebx );
    sub( 4, ebx );
    str.setstr( '-', horz, ebx );
    if( this.fillShape ) then

        add( 2, ebx );
        str.setstr( ' ', spcs, ebx );

    else

        str.cpy( "", spcs );

    endif;

    // Okay, draw the top line.

    mov( this.x, ax );
    add( 2, ax );
    console.gotoxy( this.y, ax );
    stdout.puts( horz );

    // Now draw the second line and the
    // as "/" and "\" with optional spaces
    // inbetween (if fillShape is true).

    mov( this.y, cx );
    inc( cx );
    console.gotoxy( cx, ax );
    stdout.puts( spcs );

    console.gotoxy( cx, this.x );
    stdout.puts( " /" );

    add( this.width, ax );
    sub( 4, ax );                   // Sub 4 because we added two above.
    console.gotoxy( cx, ax );
    stdout.puts( "\ " );


    // Okay, now draw the bottom line:

    mov( this.x, ax );
```

```
            add( 2, ax );
            mov( this.y, cx );
            add( this.height, cx );
            dec( cx );
            console.gotoxy( cx, ax );
            stdout.puts( horz );

            // And draw the second from the bottom
            // line as "\" and "/" with optional
            // spaces inbetween (depending on fillShape)

            dec( cx );
            console.gotoxy( cx, this.x );
            stdout.puts( spcs );

            console.gotoxy( cx, this.x );
            stdout.puts( " \" );

            mov( this.x, ax );
            add( this.width, ax );
            sub( 2, ax );                   // Sub 4 because we added two above.
            console.gotoxy( cx, ax );
            stdout.puts( "/ " );

            // Finally, draw all the lines inbetween the
            // top two and bottom two lines.

            mov( this.y, cx );
            mov( this.height, bx );
            add( cx, bx );
            add( 2, cx );
            sub( 2, bx );
            mov( this.x, ax);
            add( this.width, ax );
            dec( ax );

            while( cx < bx) do

                console.gotoxy( cx, this.x );
                stdout.putc( '|' );
                stdout.puts( spcs );
                console.gotoxy( cx, ax );
                stdout.putc( '|' );
                inc( cx );

            endwhile;

        endif;

        pop( edx );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end draw;



    /*********************/
    /*                   */
    /* Diamond's methods */
```



```
/*                    */
/*********************/


// Constructor for a diamond shape.
// See pertinent comments for the rect constructor
// for more details.

procedure diamond.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = NULL ) then

        mov( malloc( @size( rect ) ), esi );

    endif;
    mov( &diamond._VMT_, this._pVMT_ );

    // Initialize fields to create a 2x2 diamond.

    sub( eax, eax );

    mov( ax, this.x );
    mov( ax, this.y );
    inc( eax );
    mov( al, this.fillShape );  // Sets fillShape to true.
    inc( eax );                 // Minimum diamond size is 2x2.
    mov( ax, this.height );
    mov( ax, this.width );

    pop( eax );
    ret();

end create;


// We have to overload the resize method for diamonds
// (unlike the other objects) because diamond shapes
// have to be symmetrical.  That is, the width and
// the height have to be the same.  This code enforces
// this restriction by setting both parameters to the
// minimum of the width/height parameters and then it
// calls shape.resize to do the dirty work.

method diamond.resize( width:uns16; height:uns16 ); @nodisplay;
begin resize;

    // Diamonds are symmetrical shapes, so the width and
    // height must be the same.  Force that here:

    push( eax );
    mov( width, ax );
    if( ax > height ) then

        mov( height, ax );

    endif;

    // Call the shape.resize method to do the actual work:
```

```
    push( eax );    // Pass the minimum value as the width.
    push( eax );    // Also pass the minimum value as the height.
    call( shape._VMT_[ @offset( shape.resize ) ] );

    pop( eax );

end resize;


// Here's the code to draw the diamond.

method diamond.draw; @nodisplay;
var
    startY: uns16;
    endY:   uns16;
    startX: uns16;
    endX:   uns16;

begin draw;

    push( eax );
    push( ebx );
    push( ecx );
    push( edx );

    if
    (#{
        cmp( this.width, 2 );
        jb true;
        cmp( this.height, 2 );
        jae false;
    }#) then

        // Special cases for small diamonds.
        // Resizing prevents most of these from ever appearing.
        // However, if someone pokes around directly in the
        // width and height fields this code will save us:

            cmp( this.width, 1 );
            ja D2x1;
            cmp( this.height, 1 );
            ja D1x2;

            // At this point we must have a 1x1 diamond

            console.gotoxy( this.y, this.x );
            stdout.putc( '+' );
            jmp SmallDiamondDone;

        D2x1:

            // Okay, we have a 2x1 (WxH) diamond here:

            console.gotoxy( this.y, this.x );
            stdout.puts( "<>" );
            jmp SmallDiamondDone;

        D1x2:

            // We have a 1x2 (WxH) diamond here:
```

```
            mov( this.y, ax );
            console.gotoxy( ax, this.x );
            stdout.putc( '^' );
            inc( ax );
            console.gotoxy( ax, this.x );
            stdout.putc( 'V' );

        SmallDiamondDone:


    else

        // Okay, we're drawing a reasonable sized diamond.
        // There is still a minor problem.  The best looking
        // diamonds always have a width and height that is an
        // even integer.  We need to do something special if
        // the height or width is odd.
        //
        //   Odd         Odd
        //  Height      Width
        //                .   <- That's a period
        //   /\          / \
        //  <  >         \ /
        //   \/           '   <- That's an apostrophe
        //
        //          Both
        //            .
        //           / \
        //          <   >
        //           \ /
        //            '
        //
        // Step one: determine if we have an odd width.  If so,
        // output the period and quote at the appropriate points.

        mov( this.width, ax );
        mov( this.y, cx );
        test( 1, al );
        if( @nz ) then

            shr( 1, ax );
            add( this.x, ax );
            console.gotoxy( cx, ax );
            stdout.putc( '.' );
            inc( cx );
            mov( cx, startY );

            add( this.height, cx );
            sub( 2, cx );
            console.gotoxy( cx, ax );
            stdout.putc( '''' );
            dec( cx );
            mov( cx, endY );

        else

            mov( this.y, ax );
            mov( ax, startY );
            add( this.height, ax );
            dec( ax );
            mov( ax, endY );
```

```
endif;

// Step two: determine if we have an odd height.  If so,
// output the less than and greater than symbols at the
// appropriate spots (in the center of the diamond).

mov( this.height, ax );
mov( this.x, cx );
test( 1, al );
if( @nz ) then

    shr( 1, ax );
    add( this.y, ax );
    console.gotoxy( ax, cx );
    stdout.putc( '<' );
    inc( cx );
    mov( cx, startX );

    // Write spaces across the center if fillShape is true.

    if( this.fillShape ) then

        lea( ebx, [ecx+1] );
        mov( this.x, dx );
        add( this.width, dx );
        dec( dx );
        while( bx < dx ) do

            stdout.putc( ' ' );
            inc( bx );

        endwhile;

    endif;

    add( this.width, cx );
    sub( 2, cx );
    console.gotoxy( ax, cx );
    stdout.putc( '>' );
    dec( cx );
    mov( cx, endX );

else

    mov( this.x, ax );
    mov( ax, startX );
    add( this.width, ax );
    dec( ax );
    mov( ax, endX );

endif;

// Step three: fill in the sides of the diamond
//
//      /\           '
//     /  \   O     / \    (or something inbetween these two).
//     \  /    R   <   >
//      \/          \ /
//                   '
// We've already drawn the points if there was an odd height
```

```
// or width, now we've just got to fill in the sides with
// "/" and "\" characters.
//
// Compute the middle two (or three) lines beginning with
// the "/" (decY) and "\" (incY) symbols:
//
// decY = (( startY + endY - 1 ) and $FFFE )/2
// incY = ( startY + endY )/2 + 1

mov( startY, ax );
add( endY, ax );
mov( ax, bx );
dec( ax );
and( $FFFE, ax ); // Force value to be even.
shr( 1, ax );

shr( 1, bx );
inc( bx );


// Fill in pairs of rows as long as we don't hit the bottom/top
// of the diamond:

while( (type int16 ax) >= (type int16 startY) ) do

    // Draw the sides on the upper half of the diamond:

    mov( startX, cx );
    mov( endX, dx );
    console.gotoxy( ax, cx );
    stdout.putc( '/' );
    if( this.fillShape ) then

        inc( cx );
        while( cx < dx ) do

            stdout.putc( ' ' );
            inc( cx );

        endwhile;

    endif;
    console.gotoxy( ax, dx );
    stdout.putc( '\' );

    // Draw the sides on the lower half of the diamond:

    mov( startX, cx );
    mov( endX, dx );
    console.gotoxy( bx, cx );
    stdout.putc( '\' );
    if( this.fillShape ) then

        inc( cx );
        while( cx < dx ) do

            stdout.putc( ' ' );
            inc( cx );

        endwhile;
```

```
                endif;
                console.gotoxy( bx, dx );
                stdout.putc( '/' );

                inc( bx );
                dec( ax );
                inc( startX );
                dec( endX );

            endwhile;

        endif;


        pop( edx );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end draw;





    end Shapes;
```

Program 10.2 Shapes.hla - The Implementation of the Shape Class

```
// This is a simple demonstration program
// that shows how to use the shape objects
// in the shape, rect, roundrect, and diamond classes.

program ASCIIDraw;
#include( "stdlib.hhf" )
#includeonce( "shapes.hhf" )

type
    pShape: pointer to shape;


// Allocate storage for various shapes:

static
    aRect1: pointer to rect;
    aRect2: pointer to rect;
    aRect3: pointer to rect;

    aRrect1: pointer to roundrect;
    aRrect2: pointer to roundrect;
    aRrect3: pointer to roundrect;

    aDiamond1: pointer to diamond;
```

```
    aDiamond2: pointer to diamond;
    aDiamond3: pointer to diamond;

    // We'll create a list of generic objects
    // in the following array in order to demonstrate
    // virual method calls and polymorphism.

    DrawList: pShape[9];

begin ASCIIDraw;

    console.cls();

    // Initialize various rectangle, roundrect, and diamond objects.
    // This code also stores pointers to each of these objects in
    // the DrawList array.

    mov( rect.create(), aRect1 ); mov( esi, DrawList[0*4] );
    mov( rect.create(), aRect2 ); mov( esi, DrawList[1*4] );
    mov( rect.create(), aRect3 ); mov( esi, DrawList[2*4] );

    mov( roundrect.create(), aRrect1 ); mov( esi, DrawList[3*4] );
    mov( roundrect.create(), aRrect2 ); mov( esi, DrawList[4*4] );
    mov( roundrect.create(), aRrect3 ); mov( esi, DrawList[5*4] );

    mov( diamond.create(), aDiamond1 ); mov( esi, DrawList[6*4] );
    mov( diamond.create(), aDiamond2 ); mov( esi, DrawList[7*4] );
    mov( diamond.create(), aDiamond3 ); mov( esi, DrawList[8*4] );


    // Size and position each of these objects:

    aRect1.resize( 10, 10 );
    aRect1.moveTo( 10, 10 );

    aRect2.resize( 10, 10 );
    aRect2.moveTo( 15, 15 );

    aRect3.resize( 10, 10 );
    aRect3.moveTo( 20, 20 );
    aRect3.fill( false );


    aRrect1.resize( 10, 10 );
    aRrect1.moveTo( 40, 10 );

    aRrect2.resize( 10, 10 );
    aRrect2.moveTo( 45, 15 );

    aRrect3.resize( 10, 10 );
    aRrect3.moveTo( 50, 20 );
    aRrect3.fill( false );


    aDiamond1.resize( 9, 9 );
    aDiamond1.moveTo( 28, 0 );

    aDiamond2.resize( 9, 9 );
    aDiamond2.moveTo( 28, 3 );

    aDiamond3.resize( 9, 9 );
```

```
        aDiamond3.moveTo( 28, 6 );
        aDiamond3.fill( false );


        // Note for the real fun, draw all of the objects
        // on the screen using the following simple loop.

        for( mov( 0, ebx ); ebx < 9; inc( ebx )) do

            DrawList.draw[ ebx*4 ]();

        endfor;

    end ASCIIDraw;
```

Program 10.3 ShapeMain.hla - The Main Program That Demonstrates Using Shape Objects

10.17 Putting It All Together

HLA's class declarations provide a powerful tool for creating object-oriented assembly language programs. Although object-oriented programming is not as popular in assembly as in high level languages, part of the reason has been the lack of assemblers that support object-oriented programming in a reasonable fashion and an even greater lack of tutorial information on object-oriented programming in assembly language.

While this chapter cannot go into great detail about the object-oriented programming paradigm (space limitations prevent this), this chapter does explain the object-oriented facilities that HLA provides and supplies several example programs that use those facilities. From here on, it's up to you to utilize these facilities in your programs and gain experience writing object oriented assembly code.

