12 HLA Classes and Object-Oriented Programming

12.1 Class Data Types

HLA supports object-oriented programming via the class data type. A class declaration takes the following form:

```
class

   << declarations >>

endclass;
```

Classes allow **const, val, var, static, readonly, storage, procedure, iterator,** and **method** declarations. In general, just about everything allowed in a program declaration section except **labels, types,** and **namespaces** are legal in a class declaration.

Unlike C++ and Object Pascal, where the class declarations are nearly identical to the record/struct declarations, HLA class declarations are noticeably different than HLA records because you supply **const, var, static,** etc., declaration sections within the class. As an example, consider the following HLA class declaration:

```
type
   SomeClass:
          class

             var
                i:int32;

             const
                pi:=3.14159;

             method incrementI;

          endclass;
```

Unlike records, you must put each declaration into an appropriate section. In particular, data fields must appear in a **static, readonly**, **storage**, or **var** section.

Note that the body of a procedure or method does not appear in the class declaration. Only prototypes (forward declarations) appear within the class definition itself. The actual procedure or method is declared elsewhere in the code.

12.2 Classes, Objects, and Object-Oriented Programming in HLA

HLA provides support for object-oriented program via classes, objects, and automatic method invocation. Indeed, supporting method calls requires HLA to violate an important design principle (that HLA generated code does not disturb values in any registers except ESP and EBP). Nevertheless, supporting object-oriented programming and automatic method calls was so important, an exception was made in this instance. More on that in a moment.

It is worthwhile to review the syntax for a class declaration. First, class declaration may only appear in a **type** section within an HLA program. You cannot define classes in the **var, static, storage**, or **readonly** sections and HLA does not allow you to create class constants[1]. Within the **type** section, a class declaration takes one of the following forms:

```
type
        baseClass:
                class
                        Declarations, including const,
                        val, var, and static sections, as
                        well as procedures, methods, and
                        macros.
                endclass;

        derivedClass:
                class inherits( baseClass )
                        Declarations, including const,
                        val, var, and static sections, as
                        well as procedure and method prototypes, and
                        macros.
                endclass;
```

Note that you may not include **type** sections or **namespace** sections in a class. Allowing **type** sections in a class creates some special problems (having to due with the possibility of nested class definitions). Name spaces are illegal because they allow **type** sections internally (and there is no real need for name spaces within a class).

Note that you may only place **procedure, iterator,** and **method** prototypes in a class definition. Procedure and method prototypes look like a forward declaration without the forward reserved word; they use the following syntax:

```
procedure procName(optional_parameters); options
method methodName(optional_parameters); options
iterator iterName( optional_parameters ); optional_external
```

`procName`, `iterName`, and `methodName` are the names you wish to assign to these program units. Note that you do *not* preface these names with the name of the class and a period.

If the procedure, iterator, or method has any parameters, they immediately following the procedure/iterator/method name enclosed in parentheses. The parentheses must not be present if there are no parameters. A semicolon immediately follows the parameters, or the procedure/method name if there are no parameters.

12.3 The THIS and SUPER Reserved Words

Within a class method, procedure, or iterator, you will often need to access one of the class fields of the current object. Upon entry into a class method or iterator, the ESI register will always be pointing at the class object's data. Upon entry into a class procedure, the ESI register will either contain NULL (if you call the class procedure directly, specifying the class name rather than an object name) or a pointer to the object's data (if you call the class procedure using an object name or object pointer name). You can use HLA's type coercion operation to access the object's data fields or call other methods in the class, e.g.,:

```
method someClass.SomeMethod;
begin SomeMethod;

    mov( (type someClass [esi]).someField, eax );
    (type someClass [esi]).someOtherMethod( eax );
```

1. Of course, you may create class variables (objects) by specifying the class type name in the var or static sections.

```
end SomeMethod;
```

Of course, you must take care not to overwrite the value passed in ESI to the method (or iterator or procedure) when using it in this fashion.

HLA offers a special reserved word, **this**, that simplifies accessing fields of the current object. The this keyword automatically expands to "(type current_object_class [esi])", so you could write the previous code thusly:

```
method someClass.SomeMethod;
begin SomeMethod;

    mov( this.someField, eax );
    this.someOtherMethod( eax );

end SomeMethod;
```

Note that calling a class function associated with any other object will load ESI with the address of that object's data; so if you make such a call within a method the current value in ESI may be replaced. Using **this** after such a call will produce undefined results:

```
method someClass.aMethod;
begin aMethod;

    someOtherObject.itsMethod( 0 );
    mov( this.someField, eax );  // Incorrect! ESI is wiped out!

end aMethod;
```

On occasion, a method may need to call the base class' version of that method in order to handle some operations done by the base class. The intent might be like the following (incorrect example):

```
method derivedClass.someFunction;
begin someFunction;

    // Attempt to call the base class' method:

    (type baseClass [esi]).someFunction();

    // Do some work specific to this class:
        .
        .
        .

end someFunction;
```

This won't work as intended. The code above will likely end up in an infinite loop because the current object's virtual method table (VMT) entry for **someFunction** points at the **derivedClass.someFunction** method. Simply coercing the type of [esi] won't change this (indeed, this is how polymorphism in object-oriented programming works). If you really want to call the base class' method, you should use the **super** keyword. The **super** keyword is similar to **this** except that it is only valid for method calls. Consider the following example:

```
method derivedClass.someFunction;
begin someFunction;
```

```
    // Attempt to call the base class' method:

    super.someFunction();

    // Do some work specific to this class:
        .
        .
        .

end someFunction;
```

The difference between **this** and **super** is that the **super** keyword loads the EDI register (which points at the VMT) with the address of the base class' VMT rather than the current classes VMT. This forces the call to the base class' method rather than to the current (derived) class' method. See the discussion of the **override** keyword later in the chapter for more details on derived and base class methods.

12.4 Class Procedure and Method Prototypes

Class procedure and method prototypes allow two options: an **@returns** clause and/or an **external** clause. The **@**`pascal, @cdecl, @stdcall, @nodisplay` **a**nd **@**`noframe` options are not allowed in the prototype. See the section on procedures for more details on the **@**`returns` and `external` clauses. The iterator only allows the `external` option.

You can also use new style procedure declarations in an HLA class to declare procedures, iterators, and macros. Here is a simple example of a class using the new style syntax:

```
type
    myClass:
        class

            proc
                classProc:procedure( i:int32 );
                classMethod:method( j:int32 );
                classIterator:iterator( k:int32 );
            endproc;

        endclass;
```

Unlike procedures and methods, if you define a macro within a class you must supply the body of the macro within the class definition.

Consider the following example of a class declaration:

```
type
    baseClass:
        class

            var
                i:int32;

            procedure create; @returns( "esi" );
            procedure geti; @returns( "eax" );
            method seti( ival:int32 ); @external;

        endclass;
```

By convention, all classes should have a class procedure named `create`. This is the constructor for the class. The create procedure should return a pointer to the class object in the ESI register, hence the `@returns( "esi" );` clause in this example.

This procedure includes two accessor functions, `geti` and `seti`, that provide access to the class variable `i`. Note that HLA classes do not support the public, private, and protected visibility options found in HLLs like C++ and Delphi. HLA's design assumes that assembly language programmers are sufficiently disciplined such that they will not access fields that should be private[1].

Of course, the class' procedures and methods must be defined at one point or another. Here are some reasonable examples of these class definitions (a full explanation will appear later):

```
procedure baseClass.create;
begin create;

    push( eax );
    if( esi = 0 ) then

        malloc( @size( baseClass ));
        mov( eax, esi );

    endif;
    mov( baseClass._VMT_, this._pVMT_ );
    pop( eax );
    ret();

end create;


procedure baseClass.geti; @nodisplay; @noframe;
begin geti;

    mov( this.i, eax );
    ret();

end geti;

method baseClass.seti( ival:int32 ); @nodisplay;
begin seti;

    push( eax );
    mov( ival, eax );
    mov( eax, this.i );
    pop( eax );

end seti;
```

These procedure and method declarations look almost like regular procedure declarations with one important difference: the class name and a period precede the procedure or method name on the first line of the procedure/method declaration. Note, however, that only the procedure or method name appears after the **begin** and **end** clauses.

Another important difference is the procedure options. Only the **@**`nodisplay`/**@**`display`**,** **@**`noalignstack`/`@alignstack`**,** and **@**`noframe`/`@frame` options are legal here (the converse

1. Actually, HLA was designed this way because far too often programmers make fields private and other programmers decide they really needed access to those fields, software engineering be damned. HLA relies upon the discipline of the programmers to stay out of trouble on this matter.

of the class procedure/method prototype definitions which only allow `external` and `@returns`). Note that class procedures, methods, and iterators do not support the **@**`pascal`, `@cdecl`, or **@**`stdcall` procedure options (they always use the Pascal calling convention).

Class procedures and methods must be defined at the same lex level and within the same scope as the class declaration. Usually class declarations are a lex level zero (i.e., inside the main program or within a unit), so the corresponding procedure and method declarations must appear at lex level zero as well. Of course, it is legal to declare a class type within some other procedure (at lex level one or higher). If you do this, the class procedure and method declarations must appear at the same level.

Note that class declarations also support the new procedure declaration syntax with a **`proc`** section. Here is the previous example using the new style procedure declarations:

```
type
    baseClass:
        class

            var
                i:int32;

            proc
                create :procedure {@returns( "esi" )};
                geti   :procedure {@returns( "eax" )};
                seti   :method( ival:int32 ); external;

        endclass;

proc
    baseClass.create: procedure;
    begin create;

        push( eax );
        if( esi = 0 ) then

            malloc( @size( baseClass ));
            mov( eax, esi );

        endif;
        mov( baseClass._VMT_, this._pVMT_ );
        pop( eax );
        ret();

    end create;


    baseClass.geti :procedure: @nodisplay @noframe;
    begin geti;

        mov( this.i, eax );
        ret();

    end geti;

    baseClass.seti :method( ival:int32 ); @nodisplay;
    begin seti;

        push( eax );
        mov( ival, eax );
        mov( eax, this.i );
```

```
        pop( eax );

    end seti;
```

12.5 Inheritance

HLA classes support inheritance using the **inherits** reserved word. Consider the following class declaration that inherits the fields from the *baseClass* declaration in the previous section:

```
derivedClass:
    class inherits( baseClass )

        var
            j:int32;
            f:real64;

    endclass;
```

This class inherits all the fields from *baseClass* and adds two new fields, `j` and `f`. This declaration is roughly equivalent to:

```
derivedClass:

        var
            i:int32;

        procedure create; @returns( "esi" );
        procedure geti; @returns( "eax" );
        method seti( ival:int32 ); @external;

        var
            j:int32;
            f:real64;

    endclass;
```

It is "roughly" equivalent because there is no need to create the `derivedClass.create` and `derivedClass.geti` procedures or the `derivedClass.seti` method. This class inherits the procedures and methods written for `baseClass` along with the field definitions.

Like records, it is possible to "override" the **var** fields of a base class in a derived class. To do this, you use the **overrides** keyword. Note that this keyword is valid only for **var** fields in a class, you may not override static objects with this keyword. Example:

```
derivedClass:
    class inherits( baseClass )

        procedure create; @returns( "esi" );
        procedure geti; @returns( "eax" );
        method seti( ival:int32 ); @external;

        var
            overrides i: dword;   // New copy of i for this class.
            j:int32;
```

```
            f:real64;

    endclass;
```

While on the subject of class **var** objects, you should be aware that class **var** objects are not (necessarily) allocated on the stack in an activation record as are local **var** variables in a **procedure**, **method**, or **iterator**. Class **var** objects are allocated in storage associated with a class object, that actual memory could be on the stack, in static memory, or on the heap.

Occasionally, you may want to override a procedure in a base class. For example, it is very common to supply a new constructor in each derived class (since the constructor may need to initialize fields in the derived class that are not present in the base class). The `override`[1] keyword tells HLA that you intend to supply a new procedure or method declaration and you do not want to call the corresponding functions in the base class. Consider the following modifications to `derivedClass` that override the `create` procedure and `seti` method:

```
derivedClass:
    class inherits( baseClass )

        var
            j:int32;
            f:real64;

        override procedure create;
        override method seti;

    endclass;
```

When you override a procedure or method, you are not allowed to specify any parameters or procedure options except the `external` option. This is because the parameters and `@returns` strings must exactly match the declarations in the base class. So even though *seti* in this derived class doesn't have an explicit parameter declared, the `ival` parameter is still required in a call to `seti`.

Of course, once you override procedures and methods in a derived class, you must provide those program units in your code. Here is an example of a section of a program that provides overridden procedures and methods along with their declarations:

```
type

    base:   class

                var
                    i:int32;

                procedure create;
                method geti;
                method seti( ival:int32 );

            endclass;


    derived:class inherits( base )

                var
```

1. Note that the syntax is `override`, not `overrides` as is used for overriding data fields. This is an unfortunate consequence of HLA's grammar.

```
                    j:int32;

                override procedure create;
                override method seti;

                method getj;
                method setj( jval:int32 );

            endclass;



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

    push( eax );
    if( esi = 0 ) then

        malloc( @size( base ));
        mov( eax, esi );

    endif;

    mov( &base._VMT_, this._pVMT_ );
    mov( 0, this.i );
    pop( eax );
    ret();

end create;

method base.geti; @nodisplay; @noframe;
begin geti;

    mov( this.i, eax );
    ret();

end geti;

method base.seti( ival:int32 ); @nodisplay;
begin seti;

    push( eax );
    mov( ival, eax );
    mov( eax, this.i );
    pop( eax );

end seti;


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

    push( eax );
    if( esi = 0 ) then

        mem.alloc( @size( base ));
        mov( eax, esi );
```

```
    endif;

    // Do any initialization done by the base class:

    call base.create;

    // Do our own specific initialization.

    mov( &derived._VMT_, this._pVMT_ );
    mov( 1, this.j );

    // Return

    pop( eax );
    ret();

end create;

method derived.seti( ival:int32 ); @nodisplay;
begin seti;

    push( eax );
    mov( ival, eax );

    // call inherited code to do whatever it does:

    (type base [esi]).seti( ival );

    // Now handle the code that we do specially.

    mov( eax, this.j );

    // Okay, return to caller.

    pop( eax );

end seti;

method derived.setj( jval:int32 ); @nodisplay;
begin setj;

    push( jval );
    pop( this.j );

end setj;

method derived.getj; @nodisplay; @noframe;
begin getj;

    mov( this.j, eax );
    ret();

end getj;
```

12.6 Abstract Methods

Sometimes you will want to create a base class as a template for other classes. You will never create instances (variables) of this base class, only instances of classes derived from this class. In object-oriented terminology, we call this an *abstract* class. Abstract classes may contain certain methods that will always be overridden in the derived classes. Hence, there is no need to actually supply the method for this base class. HLA, however, always checks to verify that you supply all methods associated with a class. Therefore, you normally have to supply some sort of method, even if it's just an empty method, to satisfy the compiler. In those instances where you really don't need such a method, this is an annoyance. HLA's *abstract methods* provide a solution to this problem.

You declare an abstract method in a class declaration as follows:

```
type
   c: class

            method absMethod( parameters: uns32 ); abstract;

         proc
            anotherAbsMethod:method( parms:uns32 ) {@returns( "eax" )};
abstract;

      endclass;
```

The abstract keyword must follow the @returns option if the @returns option is present. In the new style procedure syntax, the abstract option must follow the declaration.

The abstract keyword tells HLA not to expect an actual method associated with this class. Instead, it is the responsibility of all classes derived from "c" to override this method. If you attempt to call an abstract method, HLA will raise an exception and abort program execution.

12.7 Classes versus Objects

An *object* is an instance of a class. In plain English, this means that a class is only a data type while an object is a variable whose type is some class type. Therefore, actual objects may be declared in the **var, static, readonly,** or **storage** declaration section. Here are a couple of typical examples:

```
var
   b: base;

static
   d: derived;
```

Each of these declarations reserves storage for all the data in the specified class type.

For reasons that will shortly become clear, most programmers use pointers to objects rather than directly declared objects. Pointer declarations look like the following:

```
var
   ptrToB: pointer to base;

static
   ptrToD: pointer to derived;
```

Of course, if you declare a pointer to an object, you will need to allocate storage for the object (call the HLA Standard Library **`mem.alloc`** routine) and initialize the pointer variable with the

address of the allocated storage. As you will soon see, the class constructor typically handles this allocation for you.

12.8 Initializing the Virtual Method Table Pointer

Whether you allocate storage for an object statically (in the **static** section), automatically (in the **var** section), or dynamically (via a call to **mem.alloc**), it is important to realize that the object is not properly initialized and must be initialized before making any method calls. Failure to do so will most likely cause your program to crash when you attempt to call a method or access other data in the class.

The first four bytes of every object contain a pointer to that object's *virtual method table*. The virtual method table, or VMT, is an array of pointers to the code for each method in the class. To help you initialize this pointer, HLA automatically adds two fields to every class you create: _VMT_ which is a static double-word entry (the significance of this being a static entry will become clear later) and _pVMT_ which is a **var** field of the class whose type is pointer to dword. _pVMT_ is where you must put a pointer to the virtual method table. The pointer value to store here is the address of the _VMT_ entry. This initialization can be done using the following statement:

```
mov( &ClassName._VMT_, ObjectName._pVMT_ );
```

ClassName represents the name of the class and ObjectName represents the name of the **static** or **var** variable object. If you've allocated storage for an object pointer using mem.alloc, you'd use code like the following:

```
mov(  ObjectPtr, ebx );
mov( &ClassName._VMT_, (type ClassName [ebx])._pVMT_ );
```

In this example, ObjectPtr represents the name of the pointer variable. ClassName still represents the name of the class type.

Typically, the initialization of the pointer to the virtual method table takes place in the class' constructor procedure (it must be a procedure, not a method!). Consider the example from the previous section:

```
procedure base.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = 0 ) then

        mem.alloc( @size( base ));
        mov( eax, esi );

    endif;

    mov( &base._VMT_, this._pVMT_ );
    mov( 0, this.i );
    pop( eax );
    ret();

end create;
```

As you can see here, this example uses the keyword this._pVMT_ rather than (type derived [esi])._pVMT_ That's because **this** is a shorthand for using the ESI register as a pointer to an object of the current class type.

12.9 Creating the Virtual Method Table

For various technical reasons (related to efficiency), HLA does not automatically create the virtual method table for you; you must explicitly tell HLA to emit the table of pointers for the virtual method table. You can do this in either the **static** or the **readonly** declaration sections. The simple way is to use a statement like the following in either the **static** or **readonly** section:

```
VMT( classname );
```

If you intend to reference a VMT outside the source file in which you declare it, you can use the **external** option to make the symbol accessible, e.g.,

```
VMT( classname ); external;
```

Note that an external declaration of this form is optional. HLA always makes the VMT name for a class an external symbol. If you actually declare the VMT (using the first declaration above), HLA also makes the VMT symbol public.

If you need to be able to access the pointers in this table, there are two ways to do this. First, you can refer to the **`classname._VMT_`** double-word variable in the class. Another way is to directly attach a label to the VMT you create using a declaration like the following:

```
vmtLabel: VMT( classname );
```

The `vmtLabel` label will be a static object of type **dword**.

As for unnamed VMT declarations, HLA will automatically make the VMT symbol (and the vmtLabel symbol) external and public. If you want to explicitly specify a named external VMT declaration, you can do so with either of the following statements:

```
vmtLabel: VMT( classname ); external;
vmtLabel: VMT( classname ); external( "externalVmtLabelName" );
```

12.10 Calling Methods and Class Procedures

Once the virtual method table of an object is properly initialized, you may call the methods and procedures of that object. The syntax is very similar to calling a standard HLA procedure except that you must prefix the procedure or method name with the object name and a period. For example, assume you have some objects with the following types (*base* is the type in the examples of the previous sections):

```
var
    b: base;
    pb: pointer to base;
```

With these variable declarations, and some code to initialize the pointers to the `base` virtual method table, the calls to the *base* procedures and methods might look like the following:

```
b.create();
b.geti();
b.seti( 5 );

pb.create();
pb.geti();
pb.seti( eax );
```

Note that HLA uses the same syntax for an object call regardless of whether the object is a pointer or a regular variable.

Whenever HLA encounters a call to an object's procedure or method, HLA emits some code that will load the address of the object into the ESI register. **This is the one place HLA emits code that modifies the value in a general-purpose register!** You must remember this and not expect to be able to pass any values to an object's procedure or methods in the ESI register. Likewise, don't expect the value in ESI to be preserved across a call to an object's procedure or method. **As you will see shortly, HLA may also emit code that modifies the EDI register as well as the ESI register.** Therefor, don't count on the value in EDI, either.

The value in ESI, upon entry into the procedure or method, is that object's **this** pointer. This pointer is necessary because the exact same object code for a procedure or method is shared by all object instances of a given class. Indeed, the **this** reserved word within a method or class procedure is really nothing more than shorthand for "(type *ClassName* [esi])".

Perhaps an obvious question is "What is the difference between a class procedure and a method?" The difference is the calling mechanism. Given an object **b**, a call to a class procedure emits a call instruction that directly calls the procedure in memory. In other words, class procedure calls are very similar to standard procedure calls with the exception that HLA emits code to load ESI with the address of the object[1]. Methods, on the other hand, are called indirectly through the virtual method table. Whenever you call a method, HLA actually emits three machine instructions: one instruction that load the address of the object into ESI, one instruction that loads the address of the virtual method table (i.e., the first four bytes of the object) into EDI, and a third instruction that calls the method indirectly through the virtual method table. For example, given the following four calls:

```
b.create();
b.geti();

pb.create();
pb.geti();
```

HLA emits the following 80x86 assembly language code:

```
lea( esi,  [ebp-12]); //b
call classname.create;

lea( esi,  [ebp-12] ); //b
mov( [esi], edi );
call( (type dword ptr [edi+geti_offset_in_VMT]); //geti

mov( [ebp-16], esi ); //pb
call  classname.create

mov( [ebp-16], esi ); //b
mov( [esi], edi );
call((type dword [edi+geti_offset_in_VMT] );  //geti
```

HLA class procedures roughly correspond to C++'s *static member functions*. HLA's methods roughly correspond to C++'s *virtual member functions*. Read the next few sections on the impact of these differences.

If you call a method within some other method using the **super** keyword, the code does not fetch the VMT pointer from the current object. Instead, the code directly loads EDI with the address of the appropriate VMT:

---

1. When calling a class procedure, HLA nevers disturbs the value in the EDI register. EDI is only tweaked when you call methods.

```
super.someMethod();
```

generates x86 code like the following:

```
lea( edi, baseClass_VMT );
call( (type dword ptr [edi+methodOffsetInVMT]));
```

12.11 Accessing VMT Fields

The VMT is basically an array of pointers. Offsets zero through `(n-1)*4`, where `n` is the number of methods in a class (including inherited methods), hold pointers to each of the methods associated with the class. The previous section described how HLA emits a call to a class method. You can manually do this by simulating the same code that HLA emits. The @offset compile-time function, when supplied with the name of a class method as its operand, will return an index into the VMT where the address of that method is found. Therefore, you could manually call a method using code like the following:

```
mov( objectPtr, esi ); // or lea( esi, objectVar );
mov( [esi], edi );     // Get VMT pointer into EDI
call( [edi+@offset( derivedClass.methodToCall )]);
```

In this example, `derivedClass` is the name of the class and `methodToCall` is the name of some method in that class. Note that you must supply the full *classname.methodname* identifier to the `@offset` compile-time function so HLA can properly identify the method. Of course, it's generally easier to call the method using `objectPtr.methodToCall`, but for those who insist on calling the method using low-level code, this is how it is done.

You might be tempted to streamline the code above to something like the following:

```
mov( objectPtr, esi ); // or lea( esi, objectVar );
call( derivedClass._VMT_[@offset( derivedClass.methodToCall )]);
```

Resist the temptation to do this at all costs! First, this defeats polymorphism; `objectPtr` might actually contain a pointer to some other class that was derived from `derivedClass`. The code immediately above will always call `derivedClass.methodToCall`, even if it actually should be calling `some_class_derived_from_derivedClass.methodToCall`. The former example will handle this correctly.

Before the super keyword was added to HLA, the accepted way to call a base class' version of some method was to manually call the method, as was done in the first example of this section (though ESI usually contained the THIS/object pointer, so you didn't normally need to load it into ESI).

In HLA v2.8 and v2.9, several new fields were added to the VMT at negative offsets from the VMT's base address. At offset -4 there is a pointer to the parent class' VMT (this field contains NULL if this is a base class that has no parent class). At offset -8 is the size, in bytes, of an object of the class' type. At offset -12 is a string object that contains the name of the class associated with the VMT. The HLA Standard Library hla.hhf header file contains a record definition you can use to access these fields in a VMT:

```
namespace hla;

    vmtRec:
        record := -12;

            vmtName     :string;
            vmtSize     :uns32;
            vmtParent   :pointer to dword;

        endrecord;
```

```
end hla;
```

Using the record definition above, you could load the class' name into EAX with a statement like this:

```
mov( (type hla.vmtRec derivedClass._VMT_).vmtName, eax );
```

Don't forget to include the *hla.hhf* header file in order to gain access to the declaration of the `vmtRec` record.

12.12Non-object Calls of Class Procedures

In addition to the difference in the calling mechanism, there is another major difference between class procedures and methods: you can call a class procedure without an associated object. To do so, you would use the class name and a period, rather than an object name and a period, in front of the class procedure's name. E.g.,

```
base.create();
```

Since there is no object here (remember, base is a type name, not a variable name, and types do not have any storage allocated for them at run-time), HLA cannot load the address of the object into the ESI register before calling create. This situation can create some big problems in your code if you attempt to use the **this** pointer within a class procedure. Remember, an instruction like "mov(this.i, eax);" really expands to "mov((type base [esi]).i, eax);" The question that should come to mind is "where is ESI pointing when one makes a non-object call to a class procedure?"

When HLA encounters a non-object call to a class procedure, HLA loads the value zero (NULL) into ESI immediately before the call. Therefore, ESI doesn't contain junk but it does contain the NULL pointer. If you attempt to dereference NULL (e.g., by accessing `this.i`) you will probably bomb the program. Therefore, to be safe, you must check the value of ESI inside your class procedures to verify that it does not contain zero.

The `base.create` constructor procedure demonstrates a great way to use class procedures to your advantage. Take another look at the code:

```
procedure base.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = 0 ) then

        mem.alloc( @size( base ));
        mov( eax, esi );

    endif;

    mov( &base._VMT_, this._pVMT_ );
    mov( 0, this.i );
    pop( eax );
    ret();

end create;
```

This code follows the standard convention for HLA constructors with respect to the value in ESI. If ESI contains zero (**NULL**), this function will allocate storage for a brand new object, initialize that object, and return a pointer to the new object in ESI[1]. On the other hand, if ESI

contains a non-null value, then this function does not allocate memory for a new object, it simply initializes the object at the address provided in ESI upon entry into the code.

Certainly, you do not want to use this trick (automatically allocating storage if ESI contains **NULL**) in all class procedures; but it's still a real good idea to check the value of ESI upon entry into every class procedure that accesses any fields using ESI or the **this** reserved word. One way to do this is to use code like the following at the beginning of each class procedure in your program:

```
if( ESI = NULL ) then

    raise( AttemptToDerefZero );

endif;
```

If this seems like too much typing, or if you are concerned about efficiency once you've debugged your program, you could write a macro like the following to solve your problem:

```
#macro ChkESI;
    #if( CheckESI )
        if( ESI = 0 ) then

            raise( AttemptToDerefZero );

        endif;
    #endif
#endmacro
```

Now all you have to do is stick an innocuous `ChkESI` macro invocation at the beginning of your class procedures (maybe on the same line as the **begin** clause to further hide it) and you're in business. By defining the boolean constant `CheckESI` to be **true** or **false** at the beginning of your code, you can control whether this "inefficient" code is generated into your programs.

12.13 Static Class Fields

There exists only one copy, shared by all objects, of any **static, readonly**, or **storage** data objects in a class. Since there is only one copy of the data, you do not access variables in the class' static section using the object name or the **this** pointer. Instead, you preface the field name with the class name and a period.

For example, consider the following class declaration that demonstrates a very common use of static variables within a class:

```
program DemoOverride;

#include( "memory.hhf" )
#include( "stdio.hhf" )
type

    CountedClass:
            class

                static
                    CreateCnt:int32 := 0;
```

1. Of course, it is the caller's responsibilty to save this pointer away into an object pointer variable upon return from the class procedure.

```
                procedure create;
                procedure DisplayCnt;

            endclass;




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

        push( eax );
        if( esi = 0 ) then

            mem.alloc( @size( base ));
            mov( eax, esi );

        endif;
        mov( &CountedClass._VMT_, this._pVMT_ );
        inc( this.CreateCnt );
        pop( eax );
        ret();

    end create;

    procedure CountedClass.DisplayCnt; @nodisplay; @noframe;
    begin DisplayCnt;

        stdout.put( "Creation Count=", CountedClass.CreateCnt, nl );
        ret();

    end DisplayCnt;



var
    b:  CountedClass;
    pb: pointer to CountedClass;

begin DemoOverride;


    CountedClass.DisplayCnt();

    b.create();
    CountedClass.DisplayCnt();

    CountedClass.create();
    mov( esi, pb );
    CountedClass.DisplayCnt();


end DemoOverride;
```

In this example, a static field (`CreateCnt`) is incremented by one for each object that is created and initialized. The `DisplayCnt` procedure prints the value of this static field. Note that `DisplayCnt` does not access any non-static fields of `CountedClass`. This is why it doesn't bother to check the value in ESI for zero.

There is a big issue with respect to static fields in a class. If you include the header file containing the class definition in more than one HLA source file (that is part of a single project), HLA will create one copy of the static object for each source file. This can produce linkage errors if you attempt to link those files together. The solution to this problem is to create an external symbol in the class declaration:

```
type

    CountedClass:
            class

                static
                    CreateCnt:int32;
                          external( "CountedClass_CreateCnt" );

                procedure create;
                procedure DisplayCnt;

            endclass;
```

The external declaration in this example expects you to provide an external int32 object named `CountedClass_CreateCnt`. You can do this (in one of the HLA source files) using code like the following:

```
static
   CreateCnt :int32; external( "CountedClass_CreateCnt" );
   CreateCnt :int32 := 0;
```

12.14 Taking the Address of Class Procedures, Iterators, and Methods

You can use the static address-of operator ("&") to obtain the memory address of a class procedure, method, or iterator by applying this operator to the class procedure/method/iterator's name with a *classname* prefix. E.g.,

```
type
   c : class
      procedure p;
      method m;
      iterator i;
   endclass;

procedure c.p; begin p;  end p;
method c.m; begin m; end m;
iterator c.i; begin i;  end i;
   .
   .
   .
   mov( &c.p, eax );
   mov( &c.m, ebx );
   mov( &c.i, ecx );
```

Please note that when you apply the address-of operator ("&") to a class procedure/method/iterator you must specify the class name, not an object name, as the prefix to the procedure/method/iterator name. That is, the following is illegal given the class definition for `c`, above:

```
static
    myClass: c;
        .
        .
        .
    mov( &myClass.p, eax );
```

12.15Program Unit Initializers and Finalizers

HLA does not automatically call an object's constructor like C++ does. There is no code associated with a unit that automatically executes to initialize that unit as in (Turbo) Pascal or Delphi. Likewise, HLA does not automatically call an object's destructor. However, HLA does provide a mechanism by which you can automatically execute initialization and shutdown code without explicitly specifying the code to execute at the beginning and end of each procedure. This is handled via the HLA `_initialize_` and `_finalize_` strings. All programs, procedures, methods, and iterators have these two predeclared string constants (**val** strings, actually) associated with them. Whenever you declare a program unit, HLA inserts these constants into the symbol table and initializes them with the empty string.

HLA expands the `_initialize_` string immediately before the first instruction it finds after the **begin** clause for a program, procedure, iterator, or method. Likewise, it expands the `_finalize_` string immediately before the **end** clause in these program units. Since, by default, these string constants hold the empty string, they usually have no effect. However, if you change the values of these constants within a declaration section, HLA emits the corresponding code at the appropriate point. Consider the following example:

```
procedure HasInitializer;
    ?_initialize_ := "mov( 0, eax );";
begin HasInitializer;

    stdout.put( "EAX = ", eax, nl );

end HasInitializer;
```

This program will print "EAX = 0000_0000" since the `_initialize_` string contains an instruction that moves zero into EAX.

Of course, the previous example is somewhat irrelevant since you could have more easily put the **mov** instruction directly into the program. The real purpose of the initialize and finalize strings in an HLA program is to allow macros and include files to slip in some initialization code. For example, consider the following macro:

```
#macro init_int32( initValue ):theVar;

    :forward( theVar );
    theVar: int32
    ?_initialize_ = _initialize_ +
                            "mov( " +
                            @string:initValue +
                            ", " +
                            @string:theVar +
                            " );";
#endmacro
```

Now consider the following procedure:

```
procedure HasInitedVars;
var
    i: init_int32( 0 );
    j: init_int32( -1 );
    k: init_int32( 1 );

begin HasInitedVars;

    stdout.put( "i=", i, " j=", j, " k=", k, nl );

end HasInitedVars;
```

The first *init_int32* macro above expands to (something like) the following code:

```
    i: forward( _1002_ );
    _1002_: int32
    ?_initialize_ := _initialize_ +
                             "mov( " +
                             "0" +
                             ", " +
                             "i" +
                            " );";
```

Note that the last statement is equivalent to:

```
?_initialize_ := _initialize_ + "mov( 0, i );"
```

Also note that the text object `_1002_` expands to "`i`".

If you take a step back from this code and look at it from a high level perspective, you can see that what it does is initialize a **var** variable by emitting a **mov** instruction that stores the macro parameter into the **var** object. This example makes use of the **forward** declaration clause in order to make a copy of the variable's name for use in the **mov** instruction. The following is a complete program that demonstrates this example (it prints "i=1", if you're wondering):

```
program InitDemo;
#include( "stdlib.hhf" )

    #macro init_int32( initVal ):theVar;

        forward( theVar );
        theVar:int32;
        ?_initialize_ :=
                _initialize_ +
                "mov( " +
                @string:initVal +
                ", " +
                @string:theVar +
                " );";
    #endmacro

var
    i:init_int32( 1 );

begin InitDemo;
```

```
    stdout.put( "i=", i, nl );

end InitDemo;
```

Note how this example uses string concatenation to append an initialization string to the end of the existing string. Although _initialize_ and _finalize_ start out as the empty string, there may be more than one initialization string required by the program. For example, consider the following modification to the code above:

```
var
    i:init_int32( 1 );
    j:init_int32( 2 );
```

The two macro invocations above produce the initialization string "mov(1, i);mov(2,j);". Had the macro not used string concatenation to attach its string to the end of the existing _initialize_ string and then only the last initialization statement would have been generated.

You can put any number of statements into an initialization string, although the compiler tools used to write HLA limit the length of the string to something less than 32,768 characters. In general, you should try to limit the length of the initialization string to something less than 4,096 characters (this includes all initialization strings concatenated together within a single procedure).

Two very useful purposes for the initialization string include automatic constructor invocation and Unit initialization code invocation. Let's consider the **unit** situation first. Associated with some unit you might have some code that you need to execute to initialize the code when the program first loads in to memory, e.g.,

```
    unit NeedsInit;
    #include( "NeedsInit.hhf" )
    static
        i:uns32;
        j:uns32;

        procedure InitThisUnit;
        begin InitThisUnit;

            mov( 0, i );
            mov( 1, j );

        end InitThisUnit;
        .
        .
        .
    end NeedsInit;
```

Now suppose that the *NeedsInit.hhf* header file contains the following lines:

```
    procedure InitThisUnit; @external;
    ?_initialize_ := _initialize_ + "InitThisUnit();";
```

When you include the header file in your main program (that uses this unit), the statement above will insert a call to the InitThisUnit procedure into the main program. Therefore, the main program will automatically call the InitThisUnit procedure without the user of this unit having to explicitly make this call.

You can use a similar approach to automatically invoke class constructors and destructors in a procedure. Consider the following program that demonstrates how this could work:

```
program InitDemo2;
#include( "stdlib.hhf" )

type
   _MyClass:
      class
         procedure create;
         var
            i: int32;

      endclass;

   #macro MyClass:theObject;
      forward( theObject );
      theObject: _MyClass;
      ?_initialize_ := _initialize_ +
                              @string:theObject +
                              ".create();"
   #endmacro


   procedure _MyClass.create;
   begin create;

      push( eax );
      if( esi = 0 ) then

         mem.alloc( @size( _MyClass ) );
         mov( eax, esi );

      endif;
      mov( &_MyClass._VMT_, this._pVMT_ );
      mov( 12345, this.i );
      pop( eax );

   end create;

   procedure UsesMyClass;
   var
      mc:MyClass;

   begin UsesMyClass;

      stdout.put( "mc.i=", mc.i, nl );

   end UsesMyClass;

static
   vmt( _MyClass );


begin InitDemo2;

   UsesMyClass();
```

```
end InitDemo2;
```

The variable declaration `mc:MyClass` inside the `UsesMyClass` procedure (effectively) expands to the following text:

```
mc: _MyClass;
?_initialize_ := _initialize_ + "mc.create();";
```

Therefore, when the `UsesMyClass` procedure executes, the first thing it does is call the constructor for the `mc/_MyClass` object. Notice that the author of the `UsesMyClass` procedure did not have to explicitly call this routine.

You can use the `_finalize_` string in a similar manner to automatically call any destructors associated with an object.

Note that if an exception occurs and you do not handle the exception within a procedure containing `_finalize_` code, the program will not execute the statements emitted by `_finalize_` (any more than the program will execute any other statements within a procedure that an exception interrupts). If you absolutely, positively, must ensure that the code calls a destructor before leaving a procedure (via an exception), then you might try the following code:

```
?_initialize_ :=
          _initialize_ +
          <<string to call constructor>> +
          "try ";

?_finalize_ :=
          _finalize_ +
          "anyexception push(eax); " +
          <<string to call destructor>> +
          "pop(eax); raise( eax ); endtry; " +
          <<string to call destructor>>;
```

This version slips a **try..endtry** block around the whole procedure. If an exception occurs, the **anyexception** handler traps it and calls the associated destructor, then re-raises the exception so the caller will handle it. If an exception does not occur, then the second call to the destructor above executes to clean up the object before control transfers back to the caller. Note that this is not a perfect solution because it does not prevent the programmer from slipping in their own **try..endtry** statement with an anyexception clause that doesn't bother to execute the *_finalize_* code.