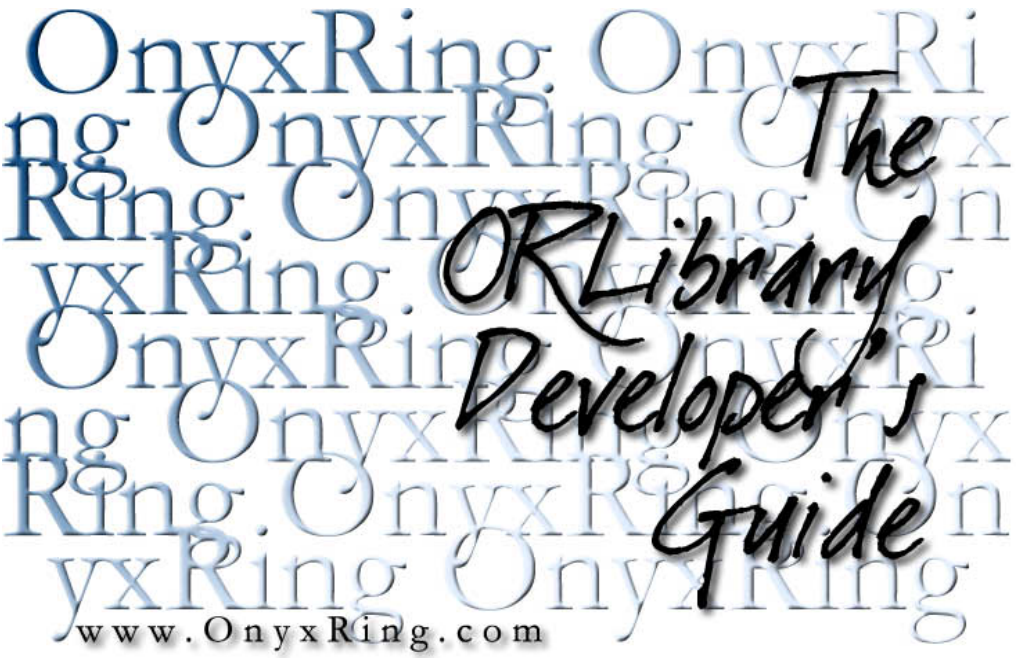

Compiled from the OnyxRing article archive on 3/12/2004 4:03:11 PM

Table Of Contents

Part A
Overview and Setup

Note:

At the time of this writing, the ORLibrary documentation is in an incomplete draft state. That is, portions of this document have not yet been written and those sections that do exist need review and revision. I can say with absolute certainty that typographical and grammatical errors abound.

The decision to release this work-in-progress was made after several requests by ORLibrary users. The library is, even by the most conservative of measures, extensive. The availability of at least some documentation greatly increases -- or so I'm told -- its usability. To that end, I am willingly undergoing the somewhat humbling experience of exposing my draft efforts to the public eye.

As new versions of this document are compiled, they will be made available from the ORLibrary's home at:

www.OnyxRing.com

Introduction

Welcome to the ORLibrary Developer's Guide. The purpose of this guide is to explain the function of the ORLibrary, give working examples of what it can accomplish, and demonstrate how it can be leveraged to create more detailed games with less effort.

This text lists the most prevalent entries in the ORLibrary. It attempts to describe each module, give examples of its use, and discuss how it interacts with other modules.

1. What is the ORLibrary?

The ORLibrary is based upon framework theory discussed in the article "Building a Personal Library" which can be found in "An Inform Developer's Guide" or online at the following URL:

www.OnyxRing.com/informguide.aspx?article=7

and also in PDF format at:

www.OnyxRing.com/downloads/aidg.pdf

It is a collection of objects, routines and enhancements that sits on top of the standard library. All the entries in the library are opt-in, meaning they have no effect whatsoever on the final produced code unless they have been selected for use. Using the ORLibrary extension framework simply makes them accessible.

2. Newbie ~=Newbie

This guide is intended to be usable by both newbie and advanced coder alike, but the term "newbie" carries a breadth of possible interpretations. First-time Informer's will have trouble getting much use from this text. If you are a newbie that has no idea what a `for` loop is, then this guide is not for you. If you have no idea what an `object` is, then this guide is not for you. If you cannot write a basic program in Inform, then this guide is not for you.

This is NOT a guide on programming in Inform. As such, it will not directly discuss topics such as "What are properties and attributes?" or "What does the Parser.h file do?" or "How do I implement a loop?" Further, it is not a tutorial on using the "standard library", so questions such as "What is the `switchable` attribute?" and "How do I change a default message?" and "What is a `before` property?" are also not directly answered. It may be that you can find answers to these questions buried in the text of this guide, but these topics are only touched upon with the assumption that you already know them or at least have a basic understanding of their theory.

Don't despair if you do not have this understanding yet. There are a number of resources at your disposal. The most notable is the Inform Developer's Manual, written by the author of the Inform language, Graham Nelson and downloadable from:

ftp://ftp.ifarchive.org/if-archive/infocom/compilers/inform6/manuals/designers_manual_4.pdf

Another helpful guide, one arguably of more use to the absolute newbie than the

DM proper, is the Inform Beginner's Guide (IBG) written by Roger Firth and Sonja Kesserich which can be found at:

www.inform-fiction.org/manual/IBG.pdf

Finally, the OnyxRing site hosts the continuously evolving manual "An Inform Developer's Guide" (mentioned above) which you may find useful as several Inform developers have had a hand in its creation. The following URL links to the site:

http://www.onyxring.com/informguide.aspx

Setting up for ORLibrary use

Setting up ORLibrary is a trivial matter. The modules can reside in virtually any directory and, with the addition of a couple of ICL settings, can be used immediately.

1. Where do I get ORLib?

The ORLibrary finds its home at the OnyxRing website. The following URL joins to the ORLibrary page:

http://www.onyxring.com/orlibrary.aspx

This page itemizes the most recent versions of each ORLibrary module, but the easiest way to download the entire library is to get the ORLib_complete.zip file, which contains a snapshot of all the entries as of the date-stamp indicated in the description.

2. Where do I install ORLib?

The best place to install the ORLibrary is entirely a matter of opinion. Whether a networked drive or your home directory, the modules can find their home anywhere that the Inform compiler can reach. Some developers simply drop the ORLibrary files in the same directory as the standard library files, saving them the initial fuss with ICL settings at the cost of code separation. This is not recommended.

It is a good idea to give the ORLibrary its own directory possibly on the same directory tier as the standard library files. The important thing to remember is that this directory will need to be added as a parameter (or switch) to the Inform compiler's command-line or ICL settings.

3. ICL Settings

There are a few ICL options that are useful when leveraging the ORLibrary. Although there are a number of ways to include ICL settings in a program, the 6.3 revision of the Inform compiler introduced support for a new ICL section in the actual source file. For simplicity, the examples given below are given in that context:

```
!% +include_path=C:\Inform\library, c:\inform\ORLibrary
!% +language_name=OREnglish
```

See the Inform Developer's Manual or the compiler's h1 option for specifics about ICL files and settings.

4a. Telling Inform where ORLibrary is installed

The first line in the above example, the "include_path" setting, is the ICL setting which tells the compiler where to search for included files. It is most often used to specify the location of the standard library files, however alternate directories can be specified when separated by a comma. For obvious reasons, this setting will vary from machine to machine and OS to OS. On a linux box, it might be specified as:

```
!% +include_path=/usr/inform/library, ~/if/inform/orlib
```

4b. Specifiying the OREnglish LDF

The second line, the settting "language_name", specifies the Language Definition File (LDF) which is used. Strictly speaking, using OREnglish is optional. Many basic library entries can be used without it, but the flexibility of this entry and its immediate transparency when implemented make it a veritable staple with the ORLibrary extensions. A few of the modules require it and raise compile-time error messages if it is not included. For simplicity, all examples in this guide assume its inclusion. OREnglish and ORPronouns (a dependency of OREnglish), will be among the first entries described in this guide.

Analysis of ORLibrary Output

Verifying that ORLibrary is installed and compiling without error is a simple matter of checking the compiler's output. We'll begin by compiling an unmodified copy of the OR_Blankgame.inf file to do this.

1. OR_Blankgame

The OR_blankgame.inf file that comes with ORLibrary has several features that make it an ideal starting point for an ORLibrary program. These will be covered more thoroughly in later sections but for now, keep in mind that it contains the two ICL settings discussed in the previous section. Compiling the file should render output similar to the following:

```
Inform 6.30 for Win32 (27th Feb 2004)
Compiling with the  O R L i b r a r y  framework
[Z-Code mode]
Preparing to process extensions REPLACE section...
          Processing library extension ORBanner...
       ---Processing the OREnglish LDF (Once only)---
          [OREnglish forcing inclusion of ORTextFormatting]
          Processing library extension ORTextFormatting...
          [OREnglish forcing inclusion of ORPronoun]
          Processing library extension ORPronoun...
Preparing to process extensions MESSAGE section...
          Processing library extension ORBanner...
          Processing library extension ORPronoun...
          Processing library extension ORTextFormatting...
Preparing to process extensions CODE section...
          Processing library extension ORBanner...
          Processing library extension ORPronoun...
          Processing library extension ORTextFormatting...
Preparing to process extensions GRAMMAR section...
          Processing library extension ORBanner...
          Processing library extension ORPronoun...
          Processing library extension ORTextFormatting...
```

If errors occur when compiling the OR_Blankgame.inf file, go back and review "Setting Up for ORLibrary Use". Most problems are usually related to directory locations and invalid attempts to signify to the compiler where files are located.

2. Verifying the Framework is Referenced

If the ORLibrary is correctly installed and functional, then the compiler output should tip you off immediately. Even without using any ORLibrary entries, the library framework alone leaves an obvious signature in the output.

2a. Checking for Framework Reference

If you are referencing the ORLibrary correctly, then right at the top of the compiler output will be the following text:

```
Compiling with the O R L i b r a r y framework
```

Additionally, the compiler will list the steps it goes through during the compilation of the ORLibrary files. This can be seen with lines that are similar to:

```
Preparing to process extensions REPLACE section...
```

The four sections that are referred to in the output will be made clear in the section entitled "Adding your own entries", but for now, just note that output from each included library module will appear four times, once for each of these steps.

2b. Checking for OREnglish

For each library module compiled, lines similar to the following are output:

```
Processing library extension ORPronoun...
```

Since the OREnglish module is a Language Definition File and not your average library entry, its output prints only once. It has also been marked to make it stand out from the other modules. To verify that the OREnglish module is indeed being compiled, simply locate the following line in the output:

```
---Processing the OREnglish LDF (Once only)---
```

2c. Automatic Dependencies

In addition to OREnglish, you may notice an unexpected side effect: three unselected library entries are pulled in. It is common practice for library entries to attempt to meet their own dependencies. That is, needed entries will often be pulled into the compilation process automatically.

After detecting that a dependency is not referenced, and forcing it to be included, an entry will then volunteer what it has done. This way, the developer is never left unaware of the source of an unexpected entry's inclusion. The following lines in the previous output example indicate this:

```
[OREnglish forcing inclusion of ORTextFormatting]
[OREnglish forcing inclusion of ORPronoun]
```

ORTextFormatting and ORPronoun are dependencies of OREnglish and a quick review of the output will show that these entries were included while processing OREnglish. An additional inclusion, ORBanner, is not really a dependency. The framework itself pulls this in to implement versioning functionality and does not produce the "forcing inclusion" line.

3. Errors with Inclusion Depth and Max Files

Prior to the release of the Inform 6.3 compiler, two errors commonly haunted ORLibrary users. The first error complicated a module's ability to meet its own dependencies:

```
Fatal error: Too many nested Includes
```

Although cumbersome, this error was usually avoided by creative coding in the ORLibrary modules themselves; not so for the second error:

```
Fatal error: Program contains too many source files:
increase #define MAX_SOURCE_FILES
```

This was a show-stopper, effectively limiting the number of ORLibrary modules that could be used in a story without modifying the compiler.

Beginning with the 6.3 compiler, two new ICL settings have introduced to address these issues. Variations of the following two ICL settings eliminate these errors:

```
!% $MAX_SOURCE_FILES=500
!% $MAX_INCLUSION_DEPTH=10
```

4. Errors with Inclusion Order

Specific rules govern where the OR_Library_Include file must be included. These are not complex rules, and are honored in the OR_BlankGame file, but they are easy to forget and break when rearranging existing source code or starting a new program from scratch.

The ORLibrary framework will detect most faulty references to OR_Library_Include and raise compile-time errors to inform you of them, such as:

```
OR_Library_Include has not been included between PARSER and VERBLIB.
```

Philosophies to keep in mind

There are a few common practices that should be kept in mind when coding. These practices are good ideas all of the time, even when not leveraging the ORLibrary, but they are especially useful when you are:

1. Layers (or base classes) by Inheritance

Using inheritance (creating derived classes) we can design layers for common changes. This practice is especially helpful when dealing with objects and yields benefits not always obvious. This is not a guide on OOP techniques but to briefly skim the surface:

- Layers allow for easy, broad changes. Changing code in a single location can affect dozens, or even hundreds, of objects. With layers, we have the ability to make sweeping changes in seconds that might otherwise take weeks.

- Layers allow for a single place to implement similarities, eliminating the need to duplicate code. Some properties and attributes are used by virtually all objects. Implementing these at a base class layer causes them to be inherited automatically by all derived objects and can shrink the size of the final sourcefile.

2. Behavior Granularity by Multiple Inheritance

I have always felt that the practice of creating objects derived from two or more classes (multiple inheritance) was an under appreciated technology. It is only by understanding and leveraging multiple inheritance that a developer can truly weld the full power OOP provides. In Inform, multiple inheritance is especially useful when designing "behaviors" -- that is, multiple classes that perform unrelated functions which are inherited by a single class or object.

One example of this can be seen in the ORNPC class and its related classes each of which encapsulate different NPC behaviors. By writing NPC behaviors in separate classes we've given the developer a granularity not otherwise available.

It is just as easy to write an NPC that walks around and talks as it is an NPC that simply talks, or one that simply walks around. More on this will be discussed in the section on NPCs "NPC Oriented Modules".

Part B
Basic ORLibrary Modules

Overview of OR_BlankGame

The OR_Blankgame.inf file, as the name implies, is a template file that can be used as a starting point for games leveraging the ORLibrary. The following is a review of various parts of the file.

1. ICL Switches

Beginning with the first line of OR_BlankGame there is a section for ICL settings:

```
!% +include_path=C:\Inform\library,c:\inform\ORLibrary
!% +language_name=OREnglish
```

These above two lines are the settings that are most often included in ORLibrary works. The first sets the compilers file search path to include the ORLibrary directory. The second, which is not requried, specifies the OREnglish language definition file. Other ICL settings can go here as well.

Note that this section is a new feature of the 6.3 compiler and ICL setting previously had to be specified either on the command line or in an ICL file.

2. Constants

With the ORLibrary framework, including registered library entries is as simple as defining the appropriate constant. In order to keep from having to lookup the appropriate constant whenever you want to include a library entry, the blank game template comes with all the current constants already defined, but commented out. All that is required to include a library entry from a file based on the OR_Blankgame.inf template is to locate the appropriate constant, and uncomment it. The following demonstrates a game including the ORDoor library entry:

```
! Specify which extensions to use
!Constant USE_ORProp;
Constant USE_ORDoor;
!Constant USE_ORSeveral;
... more constant definitions to follow this
```

3. Commented segments

There are several commented sections used to indicate where certain types of code should go. Note that this is mostly a matter of preference. Feel free to move these about or eliminate them entirely if you so desire. Honoring the convention does have two added benefits, however: It helps to keep your code organized, and it insures code is placed appropriately with respect to both the ORLibrary includes and the standard library includes.

4. Include locations

By their existance, the three standard library include files (Parser, Verblib, and Grammar) dictate four sections in any Inform program where blocks of code can be placed. These locations are:

1) Before including Parser
2) Between the inclusion of Parser and VerbLib
3) Between the inclusion of VerbLib and Grammar
4) After including Grammar

As required in every work that leverages the ORLibrary, the OR_BlankGame file hosts the following include directive In each of these four locations you will find the following include directive in each of the four required places:

```
#Include "OR_Library_Include.h";
```

OR_Library_Included is the common entry point for all framework entries. It expects to be included exactly four times and keeps track of how many times it has been included. This information is then passed along to the library entries, which may or may not choose to provide code for compilation during that pass. **All library entries must be registered with this file in order to be accessible.**

5. Additional pieces

Additionally, there are a few places in the template where generic every-game code has been created. These include the Serial and Release constants as well as the Initialize routine. These are no different than they would be in a standard, non-ORLibrary game, and are included simply for convenience.

Building a Two Room Example

We could jump immediately into an overview about the ORLibrary, but it will be much more beneficial if we have a context for our discussion. Let's begin with a "Hello World" caliber game using the OR_Blankgame template. Don't bother uncommenting any of the library entries for now. We'll get to those in a moment.

1. Beginning Layers

As was stated previously, it is a good idea to create "base classes" for our objects to derive from. This is good practice even if you have no code in the base class since it provides a place to implement common expansion in future enhancements. The following code, placed in the "Object Templates" section of the blank game template, will create a base class for objects:

```
class MyObject !---the basic object
;
```

And the same can be done for a base room object:

```
class MyRoom !---a basic room object
   class MyObject
   has light
;
```

Notice the definition of the light attribute. This is one small example of the usefulness of base classes. All rooms derived from `MyRoom` will have the `light` attribute by default. This is a useful practice since, in a hundred-room game, you've saved yourself from having to specify the light attribute 99 times. Even in our short example we see some benefit. Base classes become much more useful as game complexity increases. Remember that if you want to create a dark room which is derived from `MyRoom` you can still unset the `light` attribute (`~light`).

2. Two Rooms and an Object

Using our base classes we can create two rooms joined in an east-west fashion and a simple object in one of the rooms:

```
MyRoom Study "The Study"
  with w_to Laboratory
  ,    initial "Your eyes take a moment to adjust to the
              dimly illuminated room."
  ,    description "This is a cozy room with plush carpeting
              and walls lined in woodwork. A doorway lies to
              the west."
;
MyRoom Laboratory "The Laboratory"
  with e_to Study
  ,    description "Drab and bare, the lab is a cold sterile
              environment, not at all comfortable but brightly
              lit. Shards of glass, possibly from broken test
              tubes litterthe floor. The study lies to the
              east."
;
MyObject Candlestick "candlestick" study
  with name 'stick'
  ,    description "The candlestick is copper, or perhaps brass.
              The metal has been polished so as to gleam in the
              room's lighting. It is heavy and hard enough to
              make a formidable weapon."
;
```

3. Positioning the player

After the final step of placing the player in a room, we are finished with our two-room example. The following line goes in the `Initialize` function:

```
location=Foyer;
```

OREnglish

Even a beginning Inform programmer may notice that the previous example is no different than a plain non-ORLibrary program. What was the point of the exercise? There were two: The first was to demonstrate OREnglish's *transparency*. The second was to provide a place to demonstrate a major feature of OREnglish: The ability to create games in different tenses and person.

1. Transparency: The Same...

Other than the use of the game template and the OREnglish file, there was nothing new introduced in the coding of the previous two-room example. So what's different about the z-file produced? Very little. The same code given above will compile without the ORLibrary. As the following transcript shows running the game results in the same default messages that occur without the ORLibrary:

```
The Laboratory
Drab and bare, the lab is a cold sterile environment,
not at all comfortable but brightly lit. Shards of
glass, possibly from broken test tubes litter the floor.
The study lies to the east.

>n
You can't go that way.

>e
Your eyes take a moment to adjust to the dimly illuminated
room.

The Study
This is a cozy room with plush carpeting and walls lined
in woodwork. A doorway lies to the west.

You can see a candlestick here.

>get stick
Taken.

>eat stick
The candlestick is plainly inedible.
```

Although no differences show in the transcript, if we look at the compiler output we see that the same library entries that we discussed in section 3 were pulled in as "automatic dependencies". The OREnglish file was compiled in too, but these modules don't appear to have any affect.

This demonstrates the "transparency" of the ORLibrary entries. That is, unless instructed to do otherwise, these modules will mimic the behavior of the standard library. Without forcing it upon us, the OREnglsh module and it's dependencies

have given us substantial flexibility that we do not have with the standard library alone.

2. Tense & Person: ...Yet different

One feature that OREnglish provides us -- with the aid of ORPronoun and ORTextFormatting -- is the ability to change the person and tense of the narrative. To demonstrate this, add the following command to the `Initialize()` routine:

```
SetPersonTense(FIRST_PERSON,PAST_TENSE);
```

Recompile and run. By typing in the same commands as we did previously, we can see that the default messages have indeed changed:

```
The Laboratory
Drab and bare, the lab is a cold sterile environment,
not at all comfortable but brightly lit. Shards of
glass, possibly from broken test tubes litter the floor.
The study lies to the east.

>n
I couldn't go that way.

>e
Your eyes take a moment to adjust to the dimly illuminated
room.

The Study
This is a cozy room with plush carpeting and walls lined
in woodwork. A doorway lies to the west.

I could see a candlestick there.

>get stick
Taken.

>eat stick
The candlestick was plainly inedible.
```

All library messages have been reworked to adapt to the current person/tense settings. At the time of this writing, person settings for the `SetPersonTense` routine include `FIRST_PERSON`, `SECOND_PERSON` (the default), and `THIRD_PERSON`. Tense settings include `PRESENT_TENSE` (also the default) and `PAST_TENSE`. These can be mixed and matched.

The additional setting of `FUTURE_TENSE` was considered for a time and has partial support, but is not fully implemented.

3. Creating rudimentary person/tense sensitive Narrative

Note that in the above transcript, only the default messages have changed. The room descriptions are still rendered exactly as we typed them, in present tense. Generally this isn't a problem because games written in past tense have descriptions that reflect this just as games written in present tense. In some cases, such as reusable library entries or a games that allow the person/tense to change dynamically, it is necessary to create text that transforms itself.

3a. ppf() and fst()

The `NarativeTense` and `NarativePerson` variables are set by the `SetPersonTense()` routine. It is possible (although not recommended) to modify your text by checking these values as in the following example:

```
print "This ";
switch(NarativeTense){
    PAST_TENSE: print "was";
    PRESENT_TENSE: print "is";
}
print " a cozy room with plush carpeting and walls
    lined in woodwork. A doorway ";
switch(NarativeTense){
    PAST_TENSE: print "lay";
    PRESENT_TENSE: print "lies";
}
print " to the west.";
```

The same can be done to achieve person sensitivity by checking the `NarativePerson` variable, however this technique makes for clumsy code. OREnglish provides two rudimentary routines to help streamline the creation of person and tense sensitive narative: `ppf` (i.e.: past, present, future) and `fst` (i.e.: first, second, third). We'll touch briefly on these two routines, but be forewarned that due to nuances in the English language, they are not as useful as they may first appear. They do, however, provide the basis for a more powerful set of routines that create morphing text. We will cover these in the section "ORPronoun".

For now, they serve our needs adequately:

```
MyRoom Study "The Study"
  with w_to Laboratory
  , initial[; print (ig)fst("My","Your")," eyes ",
         (ig)ppf("took","take")," a moment to adjust
         to the dimly illuminated room.";]
  , description[; print "This ",(string) IS__TX," a
         cozy room with plush carpeting and walls lined
         in woodwork. A doorway ",(ig)ppf("lay","lies"),
         " to the west. ";]
```

```
;
MyRoom Laboratory "The Laboratory"
  with e_to Study
  , description [; print "Drab and bare, the lab ",
         (string) IS__TX," a cold sterile environment, not
         at all comfortable but brightly lit. Shards of
         glass, possibly from broken test tubes",
         (ig)ppf("littered","litter")," the floor. The
         study ",(ig)ppf("lay","lies")," to the east. ";]
;
MyObject Candlestick "candlestick" study
  with name 'stick'
  , description[; print "The candlestick ",(string) IS__TX,
         " copper, or perhaps brass. The metal ",
         (ig)ppf("had","has")," been polished so as to gleam
         in the room's lighting. It ",(string) IS__TX," heavy
         and hard enough to make a formidable weapon. ";]
;
```

Now the descriptions of the objects will be appropriate regardless of the current settings of the game-narrative:

```
The Laboratory
Drab and bare, the lab was a cold sterile environment,
not at all comfortable but brightly lit. Shards of
glass, possibly from broken test tubes littered the floor.
The study lay to the east.

>n
I couldn't go that way.

>e
My eyes took a moment to adjust to the dimly illuminated
room.

The Study
This was a cozy room with plush carpeting and walls lined
in woodwork. A doorway lay to the west.

I could see a candlestick there.

>get stick
Taken.

>eat stick
The candlestick was plainly inedible.
```

Note the use of the `(ig)` print rule when using the `ppf()` and `fst()` routines in a print statement to keep from printing the return value. This, and several other useful print rules are defined in the "ORTextFormatting" entry.

4. Changes made by ChangePlayerTense

In the above example, the reader may have noticed the use of the little-used `IS__TX` constant. Actually, the ORLibrary has redefined this into a variable and changes it appropriately ever the ChangePlayerTense() routine is called. The same has been done for the following former-constants:

ARE__TX

IS2__TX
ARE2__TX
FORMER__TX
YOURSELF__TX
CANTGO__TX

3.5 Revamped OREnglish library functions and defines

Several routines defined in the standard library's English file have been modified to be person/tense sensitive. These are:

CThatorThose
ThatorThose
ItorThem
CTheyreorThats
IsorAre

For the sake of completeness a lower-case version of the standard library's `CTheyreorThats` routine has also been implemented:

TheyreorThats

3.6 NPC Action Support

OREnglish also contains support for NPC actions. When used appropriately, standard library messages such as:

```
You put the candlestick on the table.
```

Are now sensitive to who is performing the action and may read as...

```
The troll puts the candlestick on the table.
```

...if an NPC is performing the action. More on this later in the section "ORNPC_DoVerb".

ORBanner

ORBanner is a simple extension to the standard library's banner routine. It identifies the version of the ORLibrary used and exposes the global variable/routine `ORBannerText` which can be defined in the program to add custom text to the banner.

The following code demonstrates defining `ORBannerText` as ether a string or a routine:

```
Constant ORBannerText " First time player's should type HELP.";
```

Or

```
[ORBannerText;
  print "This is a BETA version of ",(italics)Story,".";
];
```

ORPronoun

ORPronoun, a prerequisite of the OREnglish module is based upon techniques discussed in the article "Pronouns on Steroids" which can be found online at the following URL:

informguide.aspx?article=11

Contrary to its name, the ORPronoun module provides functionality to handle more than just pronouns. It implements routines that: facilitate subject-verb agreement, unify text describing NPC and player actions, and adapt a narrative to changing tenses.

1. Absolute Pronoun Print Rules

ORProunoun provides several basic print rules to implement the various types of pronouns used in the English language. These are easier to remember than more descriptive names such the `PronounAcc` or `PronounNom` examples from x62 or the DM4, since they match the first-person pronouns. The five lowercase versions of these are `(I)`, `(Me)`, `(My)`, `(Mine)`, and `(Myself)`. Similarly, the five uppercase versions are `(CI)`, `(CMe)`, `(CMy)`, `(CMine)`, and `(CMyself)`.

These ten print rules are especially useful when dealing with animate objects (although they deal equally well with non-animate objects). They utilize the `fst()` routine (OREnglish) to correctly determine the pronoun when the player is specified, but also check for gender and plurality. Individually, each print rule will produce one of eight possibilities. The `(Myself)` print rule, for example, will consult the appropriate variables and select the correct word(s) from the following list:

myself
ourselves
yourself
yourselves
their selves
itself
herself
himself

2. Potential Pronoun Print Rules

The ten absolute print rules discussed above will always print the appropriate pronoun, but there are times that we do not know for certain we should print the pronoun. What we need is an algorithm to determine if a pronoun will be understood or if the actual name needs to be used for clarity. This algorithm is implemented in the "TheXXX" print rules: `(TheI)`, `(TheMe)`, `(TheMy)`, `(TheMine)`, and `(TheMyself)`. Additionally, these five print rules have capitalized counterparts: `(CTheI)`, `(CTheMe)`, `(CTheMy)`, `(CTheMine)`, and `(CTheMyself)`.

Consider the following code snippet, modified from examples given in the " Pronouns on Steroids" article mentioned previously:

```
MyRoom foyer "Foyer"
  with description[; print "Of striking elegance, the
         foyer ",(string)IS__TX, " exquisite in every
         detail. ";]
  , each_turn[; print"^";
         if(parent(butler)~=self || parent(maid)~=self) return;
         ClearPronoun();
         if(random(2)==1) print (CTheI) butler,
              " coughs. ";
         if(random(2)==1) print (CTheI) maid,
              " scratches ",(my)maid," chin. ";
         if(random(2)==1) print (CTheI) butler,
              " grimaces at ",(theMe)maid,". ";
         if(random(2)==1) print (CTheI) maid,
              " gives ",(theMe)butler," a wink.";
  ]
;
MyObject -> butler "butler" has animate
  with name 'man'
  , description [; print (CTheI)self,
              " ignores ",(me)player,", preoccupied with
              other things.";
         ]
;
MyObject -> maid "cleaner" has animate
  with name 'maid'
  , description [; print (CTheI)maid," seems to be
              paying more attention to ",(the)self,".";
         ]
;
```

Observe the possible confusion with two male NPCs. Run the example to see how using the potential pronouns, rather than the absolute pronouns, avoids this confusion. Any of the following may be output from the random code:

```
The cleaner scratches his chin. The butler grimaces at him.
```

```
The butler coughs. He grimaces at the cleaner. The cleaner
gives him a wink.
```

```
The cleaner scratches his chin. The butler grimaces at him.
He gives the butler a wink.
```

The print rules keep track of whom the pronoun currently refers to avoid statements like "He grimaces at him". The rules are also smart enough to recognize circumstances in which pronouns can be clearly used to reference two objects. Suppose the cleaner was female:

```
MyObject -> maid "cleaning lady"
  has animate female
  with name 'maid'
  , description [; print (ig)CIVerb(self,"seemed","seem"),
         " to be paying more attention to ",
         (the)butler,".";]
;
```

The following would then be entirely possible (odds: 1/16):

```
The butler coughs. The cleaning lady scratches her chin. He grimaces at
her. She gives him a wink.
```

3. Noun-Verb Agreement

But for a single nuance in the English language, all would be well with the `fst()` and `ppf()` routines. This nuance has to do with noun-verb agreement and occurs only when three conditions are true:

1) We write in prent tense.
2) We refer to a noun in the third-person.
3) The noun is singular.

When all of the above are true -- as they are in almost all IF games with NPCs -- then we slightly modify the verb, usually by adding an 's' to the end. For instance:

You **swim** in the water.
The troll **swims** in the water.

Note the modified form of the verb "swim" in the second sentence, which matches the three conditions listed above. To describe character actions, we must first these rules to see what form the verb must take.

3a. vrb()

The `vrb()` routine assists in achieving noun-verb agreement. The first parameter expected is the object that is performing the action. This is required to check for the three conditions described above and select the verb appropriately.

The second and third parameters are the past tense and present tense forms of the verb being processed. For most verbs, these three parameters are all that are needed. Consider the following code snip which describes an action performed by either a troll or the player chosen at random:

```
Object troll "troll" has animate;

 ...

o=random(troll,player);
print (TheI)o," ",(ig)vrb(o, "swam","swim"),
     " in the water.";
```

In the majority of IF games - which are, present tense/second person - fifty percent of the time, the output will be:

```
You swim in the water.
```

And the other fifty percent of the time it will be:

```
The troll swims in the water.
```

In the second case, the `vrb()` routine has added an 's' to the verb in order to make it agree with the noun. This is all that is needed for most verbs, but some do not follow this form. "Fly," for example, will incorrectly generate the word "flys". The fourth, optional parameter will be used instead if supplied. Changing the above example, to use the verb "fly" results in the following code:

```
print (TheI)o," ",(ig)vrb(o, "flew","fly","flies"),
" in the water.";
```

And outputs the following:

You **fly** in the water.

Or:

The troll **flies** in the water.

Of course you can specify the third parameter all the time, however it is redundant since:

```
print vrb(o,"swam","swim","swims");
```

and

```
print vrb(o,"swam","swim");
```

are equivalent.

There is also a fifth parameter used to accommodate future tense as well. Like the fourth parameter, this too is optional and the `vrb()` routine will attempt to generate the correct form of the word if it is not specified. When run under a future tense context, the same example will generate a somewhat prophetic statement such as:

The troll **will fly** in the water.

It is important to note that the `vrb()` routine is not a print rule. Like the `ppf()` and `fst()` routines, when `vrb()` is embedded within a `print` statement it is usually preceeded by the `(ig)` print rule (discussed in "ORTextFormatting").

3b. Other Basic Noun-Verb Agreement Routines (am, can, have)

Additional routines that have been included because of their usefulness when generating adaptive narrative. The `(am)` print rule, for example, will print the

appropriate form of the verb "is":

was
am
will
were
are
is

Additionally, because of the frequency of their use, the print rules `(can)` and `(have)` have also been implemented.

4. Complete Noun-Verb Agreement Routines (IVerb, IAm, ICan, IHave)

Although there are times when the previously discussed routines are used alone, there is an additional set of routines that are used more frequently. Because it is more common (in English) to have a verb immediately follow the noun, the `IVerb()`, `(IAm)`, `(ICan)`, and `(IHave)` routines were created. These routines do exactly what `vrb()`, `(am)`, `(can)`, and `(have)` do but also precede the verb output with a call to the `(TheI)` print rule and a space. Therefore, the following two code snips are equivalent:

```
print (TheI)o," ",(ig)vrb(o, "flew","fly","flies");
IVerb(o, "flew","fly","flies");
```

Capitalized versions of these routines have also been created. These are: `CIVerb()`, `(CIAm)`, `(CICan)`, `(CIHave)`.

Also, because the negative form of "can" is a single word in present tense ("cannot") and two words in past tense ("could not"), two additional rules have also been supplied: `(ICant)` and `(CICant)`;

In our maid/butler example, the text is written in present tense. Using these routines, we can more easily make the example print correctly regardless of the narrative's tense and person settings and the correct verb form is correctly chosen:

```
MyRoom foyer "Foyer"
  with description[; print "Of striking elegance,
            the foyer",(string)IS__TX, " exquisite in
            every detail. ";]
  , each_turn[;
```

```
            if(parent(butler)~=self || parent(maid)~=self) return;
            print"^";
            ClearPronoun();
            if(random(2)==1) print (ig)CIVerb(butler,
                 "coughed","cough"),". ";
            if(random(2)==1) print (ig)CIVerb(maid,
                 "scratched", "scratch","scratches"),
                 " ",(my)maid," chin. ";
            if(random(2)==1) print (ig)CIVerb(butler,
                 "grimaced","grimace")," at ",
                 (theMe)maid,". ";
            if(random(2)==1) print (ig)CIVerb(maid,
                 "gave","give")," ",(theMe)butler,
                 " a wink.";
      ]
;
MyObject -> butler "butler"
   has animate
   with name 'man'
   , description [; print (ig)CIVerb(butler,
             "ignored","ignore")," ",(me)player,",
              preoccupied with other things.";
    ]
;
MyObject -> maid "cleaning lady"
   has animate female
   with name 'maid'
   , description [; print (ig)CIVerb(self,
             "seemed","seem")," to be paying more
              attention to ", (the)butler,".";
    ];
```

Wary reader's may notice the `(ig)` print rule. We will cover that in the section entitled "ORTextFormatting".

ORTextFormatting

The ORTextFormating module contains a collection of print rules that help format text.

1. (ig) ignore print rule

The ignore print rule `(ig)` is used to suppress the printing of routine return values. Since all routines have a return value, embedding them in a `print` statement causes their return value to be printed as well. Consider the first potential line of text from the `each_turn` routine in the previous example without the `(ig)` print rule:

```
print CIVerb(butler, "coughed","cough"),". ";
```

This will result in the following output:

```
The butler coughed0.
```

Notice the unwanted zero printed at the end. The `(ig)` print rule suppresses this.

At this point I feel compelled to give due credit. The `(ig)` print rule was not so much inspired by, as blatantly stolen from Roger Firth's article on print rules found at the following URL:

http://www.onyxring.com/informguide.aspx?article=8

2. italics, strong, and highlight style print rules

The Inform language has several "style" commands that change the way text is formatted to the screen. The following will print out a line of text using these different styles:

```
print "To ";
style bold;
print "boldly";
style roman;
print " go where ";
style underline;
print "no man";
```

```
style roman;
print" has gone ";
style reverse;
print "before...";
style roman;
```

The `(italics)`, `(strong)`, and `(highlight)` print rules allow formatting to be contained in a single print statement. The above code block is equivalent to the following:

```
print "To ",(strong)"boldly"," go where ",(italics)"no man",
    " has gone ",(highlight)"before...";
```

Both examples produce the same results:

To **boldly** go where *no man* has gone before...

It should also be noted that not all interpreters are equal. The `(italics)` print rule uses the "underline" style because it appears as italics on interpreters familiar to the author. Be advised, however, that other interpreters will render text printed with this print rule as underlined text which is, arguably, more appropriate.

3. (arraystring) character array print rule

The `(arraystring)` print rule has been introduced to help print the contents a character arrays. The following example, which compiles only as Z-code, will populate an array string, and then print it with the print rule.

```
array buffer -> 1000;
@output_stream 3 buffer; !--capture text
print "Hello world!";
@output_stream -3; !--release text
print (arraystring) buffer; !--now print the contents of buffer
```

The ORDynaString module leverages this print rule and makes the need to use it alone negligible.

Part C
Miscellaneous ORLibrary Modules

Canonically, the REVIEW command acts like a meta version of LOOK. That is, it enables the player to review the room description without the passages of time. This module implements is a slightly expanded version of REVIEW in that it also works for objects that have already been examined. The following transcript, occuring without the number of moves increasing, demonstrates:

>REVIEW Room A basic room. You can see a table and chair here. >REVIEW TABLE The wooden table is old and stained. >REVIEW CHAIR You have not already examined the chair.

ORBracketParserMsgs

It is a recent but well received practice to enclose parser messages in brackets. This increasingly common feature in modern games distinguishes between story narative and parser responses.

Graham Nelson speaks of the different roles of the player, the protagonist and the narrator in his article "A Triangle of Identities" (found, among other places, in the DM4). The parser is essentially an unrecognized fourth identity. It differs from the narrator in that it is concerned only with the mechanics of the game. Messages from the parser seldom add to the story itself and mostly serve to clarify commands made. Statements made by the parser are usually directed at the player behind the keyboard, rather than the player character within the game.

ORBracketParserMsgs, which depends upon the OREnglish LDF, enables this distinction. Simply including this module will wrap all parser generated messages in [brackets]. To turn off bracketing in the middle of a game, the global variable `BracketedParserMsgs` can be set to `false`.

The values of `BracketOpen` and `BracketClose` determine what text the messages will be wrapped in. These can be defined as either string values or routines. The following example, showing both of these methods, will wrap parser messages in double greater-than/less-than characters instead of the default brackets:

```
Constant BracketOpen="<<";
[BracketClose; print ">>";];
```

```
Lilly Pad
The pad floats in water.

A frog sits here, waiting patiently for bugs.

>frig, jump in water
<<You seem to want to talk to someone, but I can't see whom.>>
```

ORCenter

Center is a flexible routine used to center text on the screen. Its simplest usage is simply:

```
Center("Chapter One");
```

At first glance, this routine seems deceptively similar to the new `centre` function introduced in Inform's standard library 6/11. The above example will simply center the words "Chapter One" on the screen. `Center` is markedly more complex than its standard library counterpart, however.

In addition to supporting GLULX, `center` also supports unformatted text. That is, if the `Center()` routine encounters a line of input that is too long to fit on a single line, it will break the text at a logical place, centering all lines appropriately. `Center` further supports embedded linefeeds, much as the print command does.

In addition to the `text` parameter, there are two parameters that can be used to affect the output of `Center()`. The first is `maxwidth`, which determines the maximum width of each line and inserts line breaks accordingly. This allows box-like sections of text to be centered. If this parameter is not given, then the screen width, as provided by the interpreter, is assumed.

The second optional parameter is the `hilight` parameter. The value, defaulted to zero, can be passed in as either a one or a two. Specifying a value of one centers the text in reverse, prefixing the centered text with non-reversed spaces. A value of two prefixes the text with reversed spaces. Generally this is most useful when printing on the status line.

For the Z-Machine `center()` routine relies upon mono-spaced fonts to achieve accurate centering calculations and will temporarily assume that mode when centering text. This is not true for GLULX which leverages stylehints.

The following code snip shows an example of using the `Center()` routine:

```
Center("^^How shall the burial rite be read?^The solemn
    song be sung?^The requiem for the loveliest dead,
    ^That ever died so young? ^^-Edgar Allan Poe ~A Paean~");
```

ORTransition

The Transition() routine generates a transitional effect. It is most often used to separate passages of stand-alone text, such as chapters, by printing separating text ("---"), prompting the user to press the space bar, and then clearing the screen. Optionally, the `text` and `skiperase` parameters can be supplied to change the separation text and avoid clearing the display, respectively.

ORFirstThoughts

One consideration that must be made when writing interactive fiction concerns the presentation of player and character knowledge. The protagontist of a story often has knowledge of something that needs to be communicated to the player. The ORFirstThoughts entry provides one mechanism for accomplishing this.

Objects derived from the ORFirstThoughts class make use of descriptions in two logical parts: The character's subjective thoughts when (s)he first examines the object and the actual description of the object.

In practice, the subjective portion of a description can both precede and follow the actual description. Think of it as a description sandwich where the objective meat is wrapped in subjective bread. Each of these three description pieces are placed into one of three properties:

firsttime
descrip
firsttimePost

When an object (or room) is first examined, the text in the `Descrip` property is output between the text of the `firsttime` and `firsttimePost` properties. By default, subsequent examinations do not include these two properties.

For players that want to re-read the character's initial first impressions, the verb REEXAMINE has been implemented and variations of EXAMINE and LOOK point to this (e.g. EXAMINE BOOK AGAIN, LOOK AGAIN AT MIRROR).

On a more technical note, the ORFirstThoughts class makes use of the `description` property. Defining this property effectively disables the ORFirstThoughts functionality for any given object.

This is a good time to underscore the usefulness of base classes. Note in the following example, has a single addition to the MyObject class will affect every other object:

```
class MyObject !the basic object
  class ORFirstThoughts
;
class MyRoom
  class MyObject
;
```

```
MyRoom Study "The Study"
  with w_to Laboratory
  , initial "My eyes took a moment to adjust to the dimly
        illuminated room."
  , firsttime "The study reminded me of my grandfather. I
        am not certain why; possibly because of the manner
        in which it was decorated. "
  , descrip "It was a cozy room with plush carpeting and walls
        lined in woodwork. An iron door lay to the west."
  , firsttimepost "It was just the sort of room that my
        grandfather would have as his own. He had always
        loved woodworking. I could almost imagine him sitting
        here, puffing on his pipe."
;
MyRoom Laboratory "The Laboratory"
  with e_to Study
  , firsttime "It was a strange room to have in a house,
        a laboratory. I wondered what sort of experiments
        were conducted in the room as I looked around sizing
        it up. "
  , descrip "Drab and bare, the lab was a cold sterile
        environment, not at all comfortable but brightly lit.
        Shards of glass, possibly from broken test tubes
        littered the floor. The iron door to the study lay to
        the east."
;
MyObject Candlestick "candlestick" study
  with name 'stick'
  , descrip "The candlestick was copper, or perhaps brass.
        The metal had been polished so as to gleam in the
        room's lighting. It was heavy and hard enough to
        make a formidable weapon."
  , firsttimepost "I hefted it, calculating the force it
        would exert across the back of someone's head. It
        could definitely serve as a murder weapon."
;
MyRoom foyer "Foyer"
  with descrip "Of striking elegance, the foyer was exquisite
        in every detail. The front door on the southern wall
        was closed and locked, preventing suspects from leaving,
        but they wandered freely throughout the rest of the
        house. To the north I could see a living room; to the
        east, a long hallway. "
  , firsttime "I looked around the entrance room and
        smirked. This house was like something out of a movie.
        Extravagance was the defining trait of this crime
        scene. "
  , firsttimepost "I wandered in amazement why the suspects
        hadn't been confined for the duration of the
        investigation. "
;
```

ORDoor

The ORDoor class, an implementation of the standard door described in section 13 of the DM4, greatly reduces the amount of required code. Given that rooms are wired to a door via their direction properties:

```
MyRoom Study "The Study"
  with w_to SteelDoor
. . .
MyRoom Laboratory "The Laboratory"
  with e_to SteelDoor
. . .
```

The following example shows the excessive code required to make a typical door (Don't do this):

```
!--A sample door without the ORDoor entry
Object SteelDoor "steel door"
  has openable door static scenery
  with name 'steel' 'door'
  , door_dir [;
     if (parent(actor) == Study)
          return w_to;
     else
          return e_to;
    ]
  , door_to [;
     if (parent(actor) == Study)
          return Laboratory;
     else
          return Study;
    ]
  , found_in Study Laboratory
;
```

Note how the traditional door requires specially tailored `door_dir`, `door_to`, and `found_in` properties in addition to mapping rooms to the door. This is not necessary using the ORDoor module since it interrogates the world map and populates the three required properties during game initialization.

Therefore, by including the ORDoor module, the same door as above can be created in one line:

```
ORDoor SteelDoor "steel door" with name 'steel' 'door';
```

ORRecogName

The `name` property is the mechanism by which the Inform parser identifies which objects the player is referring to. The separation between the name and the display name has often been a source of grumbling by those who program in Inform. Many developers are annoyed at having to specify the words in the display name in the name property as well:

```
Object DirtyBook "old black leather bound book"
  with name 'old' 'black' 'leather' 'bound' 'book'
;
```

The ORRecogName library entry eliminates the need to specify the name property at all. The words of the display name will be used to identify an object and the name property can be used to specify synonyms alone.

There is no code needed to make the ORRecogName entry to work; it simply needs to be included. In the following example, notice that without this entry the `Candlestick` object can only be referenced by the word "stick":

```
object Candlestick "candlestick" study
  with name 'stick'
  , description "The candlestick was copper."
;
```

After adding this entry, all objects can also be refered to by any word that appears in their display name, making commands such as EXAMINE CANDLESTICK work here.

ORLookRoom

This module adds the ability to examine a neighboring room. That is, a player presented with the description of a room:

```
You are in the Dining Room. To the North is the Laboratory.
```

will have the ability to use any of the following:

```
>look north
>look at laboratory
>examine laboratory
```

The description of a neighboring room is determined by the following logic:

1) If the current room has a direction-description property defined, then that is used. The following are direction-description properties:

n_look
s_look
e_look
w_look
nw_look
sw_look
ne_look
se_look
u_look
d_look

2) Failing the above, the standard library's `compass_look` routine is called for the current location if it has been provided.

3) The room being looked at is searched for the `remote_description` property. If found, then that is use.

4) Alternately, the room being looked at is checked for the property

`describe_as_if_present` to be set to true. If so, then the normal room descriptions are called as though the character were actually present in the room.

Additionally, for completeness, the commands EXAMINE ROOM and LOOK AROUND have been implemented along with minor variations.

Note that determining the description of a remote location is moderately complex and an effort is made to determine if the room can actually be seen. That is, if the room is directly adjacent to the current location, or connected via a door that is either open or transparent. An additional property `visible_from` can be defined for a location to list other rooms from which it can be seen, even if it is not directly connected to them.

If the player attempts to look in a direction for which scenery text cannot be determined, then the default error message is printed. It is possible to redefine the text for this message in the traditional fashion using the LibraryMessages object, but it is often preferred to override this with room that is specific to certian locations. In these cases, the `cant_look` property may be defined.

The following code snip demonstrates this entry:

```
object mountain "mountain"
   has light
   with name 'mountain'
   ,    description "You are high on a mountain.
           Far below, to the south, lies a field."
   ,    remote_description "Far above rises the
           imposing mountain.  Snow capping the top."
   ,    w_to field
   ,    d_to field
;
object field "field"
   has light
   with name 'field'
   ,    description "It is bright and cheery here. A
           mountain lies off to the east, and a small
           open house lays to the west."
   ,    w_look "The aged house is open and
           inviting."
   ,    w_to house
   ,    e_to mountain
   ,    u_to mountain
;
object house "house"
   has light
   with name 'house'
   ,    description "It is a dark and empty house. The
           doorway to the west is open, revealing a
           field."
   ,    e_to field
   ,    e_look "The field to the east is bright and cheery.
           A mountain rises up above the horizon."
   ,    cant_look "You can't see through walls. Only
           through the open doorway to the east can
           you see anything."
;
```

In the above example, the field and the house are visible from each other by virtue of the `w_look` and `e_look` properties. The descriptions are specific to the room they are in and so it is acceptable to put directions in them ("...the field to the east...") The mountain location defines the `remote_description` property. Since the text of this property will be printed from any location that can see the mountain, it is more generalized.

Note that while the mountain is automatically visible from the field (since the locations are connected) it had to be *made* visible from the non-adjacent location (the house) with the visible_from property.

ORReview

Canonically, the REVIEW command has acted as a `meta` version of LOOK. That is, it enables the player to review the room description without the passages of time. This module implements is a slightly expanded version of REVIEW in that it also works for objects that have already been examined.

The following transcript, occuring without the number of moves increasing, demonstrates:

```
>REVIEW
Room
A basic room.
You can see a table and a chair here.

>REVIEW TABLE
The wooden table is old and stained.

>REVIEW CHAIR
You have not already examined the chair.
```

ORSupporterContainer

Inform's standard library falls short of the proverbial mark with regard to the concept of "contianers". In particular, an object can either support objects that are put on it (like a table) or contain objects that are put in it (like a bowl), but not both. This is limiting because in real life, many containers can do both. A filing cabinet, for example, might have files in it, but a stapler on it.

The ORSupporterContainer library entry implements wide-spread changes to the standard library which allow the `container` and `supporter` attributes to exist simultaneously on an object. A new attribute, `contained`, is given to objects that are "in" a container. Absence of this attribute means that the child object is "on" its supporter parent. No specialized classes need to be leveraged to achieve this new behavior, simply include the module and it will work.

To demonstrate this module in action, let's create a refrigerator and place some items both inside and on top of it:

```
class MyBook
   with short_name "leather bound book"
   ,    plural "leather bound books"
   ,    description "It was a single book, bound in leather."
   ,    name 'book' 'books//p'
;
object fridge "refrigerator" house
   has static openable supporter container
   with description "It is a small, dorm-sized refrigerator that
           seemed to double as a end table."
   ,    name 'fridge' 'refrigerator' 'table' 'dorm-sized'
;
object ->-> apple "apple"
   has edible contained
   with description "Yum! The apple looks delicious."
   ,    name 'apple'
;
MyBook ->-> book1;
MyBook ->-> book2;
MyBook ->-> book3;
MyBook ->-> book4;
```

Notice in the sample transcript how the distinction is made between what is contained in the refrigerator and what in on top of the refrigerator:

```
house
It is a dark and empty house. The doorway to the
west is open, revealing a field.

You can see a refrigerator (closed and on which
are four leather bound books) here.

>open fridge
You open the refrigerator, revealing an apple.
```

```
>search fridge
On the refrigerator are four leather bound books.
In the refrigerator is an apple.
```

ORSeveral

This module allows multiple objects of the same class to be described with an adjective rather than an exact number. For instance, it may be preferable that the description of 22 gold coins read as "several gold coins."

Two properties and one attribute have been introduced to implement this behavior:

`plural_many` - Set this property to the actual text to be displayed instead of the number. If this property is not defined for the class of the multiple objects, then the default library behavior takes precedence

`many_number` - Set this property to the number which must be surpassed in order for the new behavior to occur. If this property is not defined for the class of the multiple objects then the default is used (3).

`specific_number` - This attribute is given to a parent object (or container) to force the old behavior. This is often given to the player so that 22 gold coins reads as such in the player's inventory (instead of "several gold coins")

To demonstrate this module, we will continue with the example from the previous section by adding the `plural_many` property to the `MyBook` class:

```
class MyBook
  class MyObject
  with short_name "leather bound book"
  , plural "leather bound books"
  , plural_many "several"
  , descrip "This is a single book, bound in leather."
;
```

We'll leave `many_number` set to the default of 3, since that seems a reasonable point to begin counting books. Having done this, SEARCHing the refrigerator will now render the description as "several leather bound books" instead of listing the exact number of "four".

There is one place, that we would like to continue to express the exact number: our inventory. To accomplish this, we need to give the player object the `specific_number` attribute in the `Initialize()` routine:

```
give player specific_number;
```

Now there will never be any doubt as to exactly how many books the player is holding.

ORExamWithContents

Usually, to view the contents of a container, LOOK IN must be used. The ORExamWithContents entry is a simple module that causes the contents of a container to be described when it is examined. It takes no additional code to implement.

Using the previous example to demonstrate this module, try examining the refrigerator without the `ORExamWithContents` module. Notice that the books sitting on top of it are mentioned in the initial description of the house, but not when the player issues the EXAMINE FRIDGE command. In order to see its contents, the command EXAMINE FRIDGE must be used. Now uncomment the USE_ORExamWithContents constant and recompile. Notice that EXAMINE FRIDGE will now display the contents.

ORUniqueMultiMessage

Several identical messages are often generated when a player performs an action that affects multiple objects. Consider the following transcript:

```
>TAKE ALL BOOKS FROM FRIDGE
leather bound book: Removed.
leather bound book: Removed.
leather bound book: Removed.
leather bound book: Removed.
```

The ORUniqueMultiMessage module will analyze the responses made by each of the commands and attempt to consolidate them. By leveraging this entry, the above example would appear as:

```
>TAKE ALL BOOKS FROM FRIDGE
4 leather bound books: Removed.
```

No additional code need be written to make this functionality occur. Simply include this module and consolidated messages will begin.

ORDistinctRead

In the standard library, the commands `READ` and `EXAMINE` are mapped to the same verb which simply displays an object's description. This module changes this behavior to make a distinction between reading and just looking at an object.

To do this, ORDistinctRead relies upon the property `read_value`. If present, then the value of this property will be displayed in response to the `READ` command. If not, then a message is printed explaining that the object "cannot be read, only examined" and the normal description is printed.

The following code snip demonstrates this entry:

```
Object ->-> newspaper "newspaper"
  with name 'paper' 'newspaper' 'headline'
  , description "A wrinkled newspaper with an imposing headline."
  , read_value "In bold black letters the headline reads, ~Martians Invade!~"
;
```

ORProp

The ORProp class eases implementation of a generic object which does not need to be referenced in the game. The following is an example:

```
ORProp -> "butterflies" has pluralname with name 'butterflies';
```

Using ORProp has two distinct advantages over specifying words in the room's `name` property. First, it allows the developer to specify the `pluralname` attribute thereby generating a plurality sensitive message such as "those don't need to be referenced in this game" rather than the standard singular version "that doesn't need to be ..."

The second advantage is that it works in conjunction with packages that utilize the `name` property for rooms (Such as ORLookRoom).

Some of the functionality of the ORProp class can also be overridden. For instance, many developers prefer to provide a description for scenery objects, even though they cannot be manipulated in anyway. ORProp will allow the `EXAMINE` command if a description is provided.

Additionally, ORProp defines a property called `message` that will override the default "...that is not significant..." message. This is particularly useful when creating objects that are viewable but unreachable.

The following code snips show possible uses of ORProp:

```
ORProp -> "glass shards"
   has pluralname
   with name 'glass' 'shards'
;
ORProp -> "carpeting"
  with name 'carpet'
  ,   description "It is red, and quite nice."
;
ORProp -> "actress"
  with name 'actress'
  ,   description "She looks quite pretty up there on stage."
  ,   message  "You can't reach her.  She's up on stage."
;
```

ORGibberish

Gibberish occurs quite a bit in the antasy genre. Indeed, there are a number of stories that could use random, word-like text, but making gibberish isn't as strainght-forward as producing random characters. The ORGibberish module supplies the ability to generate random, human-pronounceable words of varied length.

The ORGibberish object defines a method called `MakeWord()`. `MakeWord()` takes a single number as a parameter which specifies the number of requested syllables. It then generates a random word, with the requested number of vowels, based upon rules derived from phonetic principals of pronunciation and then prints it.

Of what possible use could this module be? Random passwords, the names of monsters, and spell incantations are a few examples.

Using the ORGibberish object is easy, although it can be leveraged in more powerful ways, as we will see in the ORDynaString section. For now, the following will demonstrate:

```
[Initialise;
    print "^";
    ORGibberish.MakeWord(2);
];
```

Run the program to see the randomly generated word. The output will likely not be even remotely similar to:

```
squeesle
```

For another example of using the ORGibberish object, refer to the ORDynaString section.

ORDynaString

Hundreds of Inform games have been written without dynamic strings. The need for strings in Inform is not as great as it is in other programming languages. There are, though, a few occasions when building a string dynamically is more elegant than alternative methods.

ORDynaString is a wrapper around the Z-machine's @output_stream functionality and the GLULX stream functionality. That is, it is turned on to print text to an array, and then turned off to restore printing to the screen. The resulting string, stored in an array pointed to by the `buf` property can then be printed using the `(dynastring)` print rule. For interchangeability, the (dynastring) print rule also works equally well with character arrays.

The following routines are exposed from the ORDynaString class:

- buf - points to an array which will store the target string.
- capture() - redirect all printed text to the array pointed to by buf.
- release() - stop redirection text to array and redirect back to the screen.
- upper(id) - takes the character stored in the array at position id (1-based index) and makes it uppercase.
- lower (id) - takes the character stored in the array at position id and makes it lowercase.
- upper_all() - makes the entire string upper case.
- lower_all() - makes the entire string lower case.
- get_char(id) - returns the character at position id.
- set_char(id,val) - sets the character at position id to the value val.
- strlen() - returns the length of the string.
- substring(start,count) - prints a portion of the total string.

There are several obscure uses for the ORDynaString class. One of these uses is the ability to maintain a copy of randomized text. The following example is an elaborate implementation of ORDynaString. Specifically, it is an office desk toy that displays a different word/definition. The word/definition changes every time it is shaken. For amusement, we'll leverage the ORGibberish entry to generate random, human-pronounceable words and piece together a random definition to accompany them. We will also leverage the `read_value` property defined in the

ORDistinctRead module.

```
ORDynaString rword;
ORDynaString rtext;
array word_buf -> 30;

Object word_cube "dictionary cube" glasstable
  with name 'word' 'definition' 'toy' 'office' 'electric'
  , description "It was an electric office-toy in the shape
         of a cube. It displayed a changing dictionary
         definition of little-known words which could be
         read at any time."
  , before[;
         Shake: self.Shuffle();
              print (ig)CIVerb(actor,"shook","shake")," the
              cube and the words ", (ig)ppf("rearranged",
              "rearrange")," themselves.";
              rtrue;
  ]
  , Shuffle[;
         give self general;  !----we have shuffled at least once
         rword.buf=word_buf; !---assign different array
                             !---so dynas don't overwrite
                             !---each other
         rword.capture(); !----start capturing word text
         ORGibberish.MakeWord(2); !---make gibberish
         rword.release(); !---done with new word.
         rtext.capture(); !---now capture formatted text
         print "On the cube appeared the following
                  word/definition pair:^^";
         rword.upper(1); !--capitalize the word
         print"~",(dynastring)rword," - ";
         rword.lower (1); !--uncapitolize
         !---generate and print a random definition...
         noun=random(true,false);
         if(noun) print "(N) The"; else print "(V) To";
         print " ",(dynastring)rword," is ";
         if(noun) print "a"; else print "to";
         if(noun){
              print (string)random("n abysmal"
                   ,"n abominable"," brainless"," dreadful"
                   ," horrendous"," miserable"," stupid"
                   ," fabulous"," marvelous","n amazing"
                   ," popular"," delightful"
                   , " valuable")," ";
              print (string)random("barrel of"
                   ,"apology for","pile of","heap of"
                   ,"sack of","glob of","paragon of"
                   ,"modal of","example of", "hunk of"
                   , "bundle of","manifestation of")," ";
              print (string)random("monkey filth"
                   ,"vomit","gunk","garbage","fertilizer"
                   ,"lizard snot","disirablity" ,"precision"
                   ,"accomplishment","pig phlegm"
                   ,"maggot fodder");
         }
         else{
              print " ";
              print (string)random("cry to"
                   ,"yell at","smell","examine closely"
                   ,"mix up","spit upon","pee against"
                   ,"insult","rub down","worship","flatter"
                   ,"propose to","forget about")," a";
              print (string)random(" stranded"
                   ," dirty"," neighbor's favorite"
                   ," free"," smelly"," disproportionate"
                   ," slightly used","n insignificant"
                   ,"n extrordianary"," stolen"
                   ," recently washed"," spoiled")," ";
              print(string)random("daughter","idol"
                   ,"donkey","tree","mailbox","monkey"
                   ,"wife","brother","lunch","coat"
                   ,"lawyer");
         }
         print ".~^^Followed by the words ~Shake cube to change
              definition.~";
         rtext.release(); !----done capturing text
```

```
  ]
  , read_value [;
        if(self hasnt general) self.shuffle();
        print(dynastring)rtext;
  ]
;
```

To complete the example we can define the verb "shake":

```
Verb "shake" * held -> Shake;
[ShakeSub;print (ig)CIVerb(actor,"shook","shake")," ",(the)noun,
  ", and ",(ig)vrb(actor,"felt","feel")," better, less tense.";];
```

The following sample transcript demonstrates:

```
>take cube
Taken.

>examine cube
It was an electric office-toy in the shape of a cube. It
displayed a changing dictionary definition of little-known
words which could be read at any time.

>read cube
On the cube appeared the following word/definition pair:

"Zaquask - (N) The zaquask is a brainless hunk of
accomplishment."

Followed by the words "Shake cube to change
definition."

>shake cube
I shook the cube and the words rearranged themselves.

>read cube
On the cube appeared the following word/definition pair:

"Akrou - (V) To akrou is to rub down a dirty daughter."

Followed by the words "Shake cube to change
definition."

>shake cube
I shook the cube and the words rearranged themselves.

>read cube
On the cube appeared the following word/definition pair:

"Rekril - (N) The rekril is a horrendous bundle of
lizard snot."

Followed by the words "Shake cube to change
definition."
>
```

ORDynaMap

This module dynamically creates the ties between certain rooms based upon movements by the player, similar to an effect in the "mars" scene of Adam Cadre's "Photopia".

The use is fairly straightforward: Simply create an instance of the ORDynaMap class with the found_in property containing a list of the rooms that are to be arranged, in the order that they are to appear. The first entry in the list should already be accessable by the player.

As for the rooms that are to be created, they should initially provide all directions. The `cant_go` property on the `ORDynaMap` object, if defined, will be propagated down to the rearranged rooms which also provide cant_go, but that do not define it.

To simplify the creation of the rooms that can be rearranged by the ORDynaMap object, the `ORDynaMapRoom` class defines the needed properties. Note the following example which allows the player to explore in virtually any direction from the water's edge:

```
ORDynaMap
  with cant_go "The trees are too dense to travel
                in that direction."
  ,   found_in wateredge sparseforest forestmidst
                  forestclearing treasuretrove
;
ORDynaMapRoom wateredge "Edge of water/forest"
  with s_to "Your boat is that way, but you can't
             go home until the treasure is found."
  , description "This is a small strip of land separating
             the southern inlet of ocean from the vast forest
             in all other directions."
;
ORDynaMapRoom sparseforest "Sparse Forest"
  with description "This is the start of a vast forest.
             The trees here are thin and sparse."
;
ORDynaMapRoom forestmidst "Midst of Forest"
  with description "Smack dab in the middle of the forest.
             Trees are everywhere."
;
ORDynaMapRoom forestclearing "Forest Clearing"
  with description "Suddenly clear of trees. You feel yourself
             draw closer to the target."
;
ORDynaMapRoom treasuretrove "Pile of Treasure"
  with description "At last! The long sought-after pile of
             treasure. You are victorious."
;
```

ORCantGoOdd

CantGoOdd is a modification to the regular `GoSub` routine which causes it to ignore the `cant_go` property if the direction is up, down, in, or out. This seems to be one of the more common oddities in Inform adventures. For example...

```
You are in a small forest, surrounded by
trees. Only to the north is there a passage.

>s
Trees block travel in that direction.

>in
Trees block travel in that direction.

>down
Trees block travel in that direction.

>up
Trees block travel in that direction.
```

Alternatively, a new property has been defined called `cant_go_odd` which will be used instead with these directions. A default library message has also been created for generic `cant_go` responses in these odd directions. (Go #13)

ORDipensor

The ORDispensor class defines an object that dispenses a seemingly endless quantity of a specific object. It is useful for objects such as a pot of gold, where the pot contains numerous gold coins, but the developer does not want to create all the coins at once.

The class of the items that are dispensed must be defined in the property `itemclass`. When the program is first run, the dispensor object automatically fills itself with a number of `itemclass` instances. This number is defined in the `initial_count` property. When the items are removed, more are generated to replace them; when the items are returned to the dispensor, they are destroyed.

The following properties are of particular interest when dealing with the ORDispensor class:

- itemclass - Set this property to the class of the array that will be dispensed.
- conceal_dispensing_items - Set this to true if the dispensing items will have the concealed attribute while they are in the object. This is particularly useful when its dispensing item defines the object represented by this class. For example, a pond is only a pond by virtual of the fact that it has water in it. In this case, a description of a pond full of water should show it as empty despite being filled with water, so this property would be set to true.
- accept_alternate_itmes This property is set to disallow insertion of items that are not of the same class as the itemclass.
- Initial_count Set this to the number of instances of itemclass that should initially be visible.
- receive_item_msg outputs the message when an item is inserted into the object.
- cannot_receive_item_msg outputs the message when an item cannot be inserted into the object.

As an example, we'll create a candy dish with candy:

```
class candy(40)
  has edible
  with short_name "piece of dry candy"
  , name 'candies//p' 'candy' 'peppermint'
  , description "It is an unwrapped peppermint. "
```

```
  , plural "dry candies"
  , before[;
        eat: candy.destroy(self);
             "You eat the piece of candy."
           ]
;
ORDispensor CandyDish "dish" coffeetable
  has open
  with name 'crystal'
  , itemclass candy
  , accept_alternate_items false
  , initial_count 10
  , description "The dish is made of crystal and full of candy. "
;
```

ORReferByContents

The ORReferByContents object provides a specialized parse_name routine that allows a container to be referred to by the objects it contains. For instance, when given a jar of marbles, it is perfectly natural for the player to try to PUT THE MARBLES ON THE TABLE when PUT JAR ON TABLE is what was intended. This is even more likely for other types of contained substances, such as a bottle of water.

By default, any object that inherits from the ORReferByContents class can be referred to by its contents, regardless of the intended action. This works well for verbs like DROP, TAKE or THROW. There are also several actions where considering the container rather than its contents may not be the appropriate solution. EAT is one such example of this. To cause the container to not inherit its contained'd object's names for specific verbs, the `ignore_actions` property list should be defined.

Alternatively, when more verbs shoudl be disqualified than allowed, the `refer_actions` property list can be defined.

The following code defines a glass bottle in which objects can be placed:

```
ORReferByContents glassbottle "glass bottle" coffeetable
  has transparent open container
  with description "It was just a glass bottle, capable of
        containing all kinds of things."
  , ignore_actions ##Eat ##Drink
;
```

Note that the ORReferByContents does not automatically imply the container attribute since it could just as easily have been a tray.

ORLiquid

Liquids have been touted as being some of the most troublesome objects to implement. This is primarily because they do not act as regular objects. Since they follow different rules, many of the standard library's verb implementations cannot be applied to unbottled liquids. You cannot `TAKE` water without it flowing through your fingers, and to `DROP` water on the floor is to have it seep into the carpet, effectively disappearing. There are also additional verbs, such as `POUR`, apply only to liquid substances. Liquids are further difficult to implement because a quantity of liquid can almost always be divided in half. There are also non-liquid substances that follow some of the same basic rules as liquid, such as sand.

Adding to the difficulty of coding liquids is the fact that they are always treated neither as singular nor plural. One does not say that they have "a water" as they would have "a coin." Neither do they say "the water are in the bowl" as you would say that the "coins are." For non-liquid substances that can be handled in collections (like coins), there is both a plural and a singular form (e.g. "coin" and "coins"). This is not the case with liquid-like substances. Although there may be several water objects defined in a game, to refer to each object alone requires a form of measurement that plurality and singularity can be applied to, such as "ounces" or "handfuls" or "measures".

The ORLiquid module was designed to ease the difficulty in implementing liquids in a game. Two classes are defined in the module: ORLiquid and ORLiquidSource which is an implementation of ORDispensor. Additionally, several liquid specific verbs have been defined , such as `SCOOP` and `POUR`.

Two routines have also been created to determine if an object can contain liquids of various types: `CanContainDryLiquid()`, and `CanContainWetLiquid()`.

Also some attributes have been created to help define the world model:

- wetliquid - to be applied to ORLiquid objects if they are to act more like water than so-called dry liquids (such as sand).
- wet - given to objects that have been dipped in wet liquids.
- water_tight given to containers so signify that they can, indeed contain wet liquids.

Additionally, because liquids cannot be poured into just any container, a property called `liquid_measures` must be provided by any container that can contain liquid. This property defines what quantity, or how many measures, of liquid can be contained

in the object. Note that this property can also be defined for other objects that are not containers as well. These objects, when placed within a liquid carrying container, reduce the amount of space available for liquid. For example, a glass with a liquid_measures property of five, that contains three ice cubes, each with a liquid_measures of one, will have room for two measures of liquid.

For an example, we'll create a bottle and make it capable of holding five measures of water:

```
MyObject glassbottle "glass bottle" coffeetable
  class ORReferByContents
  has transparent open container water_tight
  with descrip "It was just a glass bottle, capable of
        containing all kinds of things."
  , ignore_actions ##Eat ##Drink
  , liquid_measures 5
;
```

Now we can define two liquid substances, one dry, and the other wet:

```
class sand(20)
  class ORLiquid
  with short_name "sand"
  , plural "handfuls of sand"
;
class water(20)
  class ORLiquid
  has wetliquid
  with short_name "water"
  , plural "ounces of water"
;
```

Nest we'll create beach and ocean objects that will dispense sand and water respectively:

```
ORLiquidSource beachsand "beach" beach
  has supporter ~container
  with itemclass sand
  , conceal_dispensing_items true
;
ORLiquidSource ocean "ocean" beach
  with itemclass water
  , conceal_dispensing_items true
;
```

ORSuppressTakeAll

There's been a fair amount of attention paid to the suppression of TAKE ALL in games. The DM4 discusses a technique for suppression of scenery objects using TAKE ALL in section 33. Roger Firth demonstrates disabling TAKE ALL completely at the following URL:

http://www.firthworks.com/roger/informfaq/aa.html#7

This module implements a similar implementation. Simply include this module and DM4's method of ignoring scenery objects from take all will be in effect. However, if the developer would rather suppress TAKE ALL entirely, then this behavior can be turned on by setting the `take_completely` property of the SuppressTakeAll object (probably best done in `Initialise`):

```
SuppressTakeAll.take_completely=true;
```

Note that this technique differs from Roger Firth's in that it still allows for qualified lists. For example, "TAKE ALL APPLES" will still function as expected.

Because TAKE ALL can take the form of the REMOVE command (e.g.: TAKE ALL FROM BOX),this command can also be suppressed in the same manner as TAKE. Similar to the `take_completely` propery, the unqualified REMOVE ALL command can be completely suppressed with the `remove_completely` property:

```
SuppressTakeAll.remove_completely=true;
```

Again, qualified REMOVEs will continue to work (e.g.: REMOVE ALL MARBLES FROM BOX).

ORBetterChoice

ORBetterChoice uses the ORLibrary's extensible entry point functionality (OREntryPoints) to implement a ChooseObjects routine. This routine adds more intelligence to the Parser's ability to decide between objects. For example, with ORBetterChoice referenced, the parser will give less priority to objects that are already carried during a TAKE action. Edible objects will be preferred to non-edible ones during the EAT command. Additionally, objects already referrenced as the subject of a sentence (noun) will be given less priority when determining the object (second).

ORExits

The `ORExits` module implements the typical EXITS verb as well as a standalone routine which does the same thing (`DescribeExits`). This functionality actually walks through and tests each of the `Compass` directions, finally generating the appropriate text when all directions have been determined.

There are times that a developer may want suppress or change this functionality for specific rooms (perhaps choosing to not describe a hidden exit). The default implementation can be overridden at the room level by defining the property `exits_text` as either a string or a routine.

ORMiniMenu

A menu system (that works for both Z Code and GLULX). This menu does not clear the screen so is ideal for CYOA style decisions that affect game play. It additionally allows for a parent/child relationship between menu items which enables the user to traverse a tree of options to locate a selection.

The following is an example of how a menu system might be implemented:

```
ORMiniMenu toplevel_m "Choose from the following";
ORMiniMenu -> "Do stuff with sticks.";
ORMiniMenu ->-> takesticks "Pick up sticks.";
ORMiniMenu ->-> burnsticks "Set sticks on fire.";
ORMiniMenu ->-> kicksticks "Kick sticks.";
ORMiniMenu -> "Look at stuff";
ORMiniMenu ->-> ash "Look closely at the fire ash.";
ORMiniMenu ->-> river "Look in the river.";
ORMiniMenu ->-> hole "Look into the hole.";
ORMiniMenu ->-> fish "Look at the fish.";
ORMiniMenu ->-> dog "Look around the tree.";
```

Calling the menu programmatically can be accomplished with a call to the "show" property which also returns the user's selection (if any):

```
result=toplevel_m.show();
```

The result is the actual menuobject. It is determined when a menu option that has no children is chosen. The user's "choice" is also stored in the result parameter of the parent menu item (that is, the menu item actually called):

```
print (name)toplevel_m.result;
```

The above code will print the user's selection (provided there was a selection. A zero result indicates no user selection (the user backed out of the menu rather than make a choice). Obviously the result should be checked for a zero value before attempting to use it.

The `menupos` property of the called menu object will determine the position of the menu at either the top of the screen or the bottom of the screen. The following are valid values for this property:

```
ORMENU_BOTTOM

ORMENU_TOP
```

Note that `ORMenu_Bottom` does not work on all interpreters. Generally it works on Frotz-based and GLK-based interpreters, but not Zip-based.

ORActionMenu

The ORActionMenu module implements a menu from which a player can choose actions. It is a system upon which so-called "Choose-Your-Own-Adventure" (CYOA) stories can be built, but differs from other CYOA systems in that it does not replace the way the world model is coded. Instead, it provides an alternative to the input and parsing routines of the standard library. All menu selections boil down to verb routines, just as they would in a traditional Inform game. This makes it possible to turn the menu system on and off, alternating between the menu and the traditional input prompt.

As with many of the sections in this guide, only the most common features of this module will be covered here. Descriptions of several the advanced features have been omitted for the sake of clarity.

Crying Baby

To provide a basis for discussion of this module, consider the following sample code:

```
!% +include_path=\inform\library,\inform\ORLibrary
!% $MAX_INCLUSION_DEPTH=10
Constant Story  "Crying Baby";
Constant Headline "^A One-Room Example^Copyright (c) 2004: Jim Fisher^";
#Include "OR_Library_Include"; #Include "Parser";
#Include "OR_Library_Include"; #Include "VerbLib"; #Include "OR_Library_Include";
object changing_room "Changing Room" has light
  with description "A small public room, used by many to tend to their babies. A
        vending machine is mounted on the wall."
  ,    cant_go "You must take care of your sister's screaming baby before leaving.";
object -> baby "baby" has animate
  with article "your sister's"
  , name 'baby' 'child' 'sister^s'
  ,    description "Your baby nephew, quite fussy and screaming loudly."
  ,    life[; give:
        if(noun==pacifier){
            deadflag=2;
            "You stick the pacifier into the baby's mouth.  He immediately stops
                crying, suddenly content.";
        } ]
  , each_turn[; "^The child screams the ear-piercing scream of an unhappy baby.";];
object -> vendingmachine "vending machine" has container scenery open
  with name 'vending' 'machine' 'slot' 'tray'
  ,    description "A simple vending machine, mounted on the wall with a slot for
        money and a tray."
  , before[; insert, puton, receive:
        if(noun==silverdollar){
            remove silverdollar;
            move pacifier to self;
            "The coin disappears down the vending machine's slot. A pacifier is
                discharged into the tray.";
        }];
object silverdollar "silver dollar" selfobj
  with    description "A round coin or silver. Worth a dollar."
  ,        name 'silver' 'dollar' 'money' 'coin';
object pacifier "pacifier"
  with description "A synthetic rubber nipple.  Exactly the sort of thing to settle a
```

```
        fussy baby."
    ,    name 'pacifier' 'rubber' 'nipple';
[Initialise;
    print "^^The baby, left with you by your sister, is clean and fed.  So why
           is he still screaming at the top of his lungs?^^^";
    location=changing_room;
];
#Include "Grammar"; #Include "OR_Library_Include"; end;
```

Although the source references the OR_Library_Include.h file, no modules are identified for inclusion. The resulting one-room game therefore plays identically to a traditional Inform story compiled without the framework. As the following transcript shows, it is winnable in three moves:

```
The baby, left with you by your sister, is clean and fed.  So why is he still
screaming at the top of his lungs?

Crying Baby
A One-Room Example
Copyright (c) 2004: Jim Fisher
Release 1 / Serial number 040227 / Inform v6.30 Library 6/11
ORLibrary release 1.2E (2004.02.27)

Changing Room
A small public room, used by many to tend to their babies. A vending machine is
mounted on the wall.

You can see your sister's baby here.

>put coin in vending machine
The coin disappears down the vending machine's slot. A pacifier is discharged into
the tray.

The child screams the ear-piercing scream of an unhappy baby.

>take pacifier
Taken.

The child screams the ear-piercing scream of an unhappy baby.

>give pacifier to baby
You stick the pacifier into the baby's mouth.  He immediately stops crying,
suddenly content.

    *** You have won ***
```

Showing the ActionsMenu

Defining USE_ORActionMenu to select the module for compilation makes the ActionsMenu command (abbreviated as AM) immediately available. It is a bit useless at this stage, though:

```
>ActionMenu
There are no actions currently in the menu system.
```

To populate the menu, use the the ActMenu object's `add` routine to supply

instances of the ORAction-derived class `ORMenuAction`. You can define ORMenuAction objects as stand-alone, but since actions are often discarded after use, it is more efficient to create dynamic instances by utilizing the predefined class array. In our example, we'll do this in `initialize()` and add the three commands that are required to complete the game:

```
[Initialise;
    print "^^The baby, left with you by your sister, is clean and fed.  So why
        is he still screaming at the top of his lungs?^^^";
    location=changing_room;
    ActMenu.add(ORMenuAction.create(##Insert, silverdollar, vendingmachine));
    ActMenu.add(ORMenuAction.create(##Take, pacifier));
    ActMenu.add(ORMenuAction.create(##Give, pacifier, baby));
];
```

Now the game will respond to the ActionsMenu command with one of three menu options with descriptions pieced together from the actions themselves. Initially:

```
                        >Insert silver dollar vending machine<
```

When a menu item is selected, command text is generated and printed as though the player typed it:

```
>Insert silver dollar vending machine
The coin disappears down the vending machine's slot. A pacifier is discharged into
the tray.
```

Customizing the menu and output

Piecing together the description from the command is a last-ditch effort by the module to describe the action. The second parameter of the `ActMenu` object's `add` method lets us specify more convincingly what we would imagine a player might type. For this first action, let's dress it up a little:

```
ActMenu.add(ORMenuAction.create(##Insert, silverdollar, vendingmachine)
  , "Put the coin into the vending machine");
```

Although this new text is tailored to print at the input prompt, the ActionMenu will assume that it is better than the generated command and also display it as a menu

option. The third parameter of the `add` method will override this as well:

```
ActMenu.add(ORMenuAction.create(##Insert, silverdollar, vendingmachine)
  ,  "Put the coin into the vending machine"
  ,  "Waste my last silver dollar in that crummy box.");
```

Filtering with CanAttempt

It's worth pointing out the obvious here. Specifically, not all of the options that we added are initially shown in the menu. This is due to ORAction's `CanAttempt` property, which is intelligent enough to know that the TAKE PACIFIER action cannot be attempted if there is no pacifier nearby. Similarly, the GIVE PACIFIER TO BABY action isn't possible. These actions, not possible until the game's state has progressed, are initially hidden.

Override the `CanAttempt` property to return `true` or `false` to change this default behavior.

PreFilterActionMenu

With the possible exception of games winnable in three turns, it is unlikely that loading all possible actions into the menu at once is practical. For most scenarios, the `PreFilterActionMenu` entry point should be defined. This entry point is called before a menu sorts out which of its options can be displayed and which cannot. It is a good place to clear out the contents of the menu and refill it with more pertinent actions.

The following shows the code, previously housed in `initialise()`, implemented in this entry point:

```
[PreFilterActionMenu;
   ActMenu.clear();
   ActMenu.add(ORMenuAction.create(##Insert, silverdollar, vendingmachine)
        , "Put the coin into the vending machine"
        , "Waste my last silver dollar in that crummy box.");
   ActMenu.add(ORMenuAction.create(##Take, pacifier));
   ActMenu.add(ORMenuAction.create(##Give, pacifier, baby),
              "Give the pacifier to the baby");
];
```

ORMenu

This is a full screen menu help system, derived from ORMiniMenu, which works for both Z Code and GLULX.

The following is an example of how a menu system might be implemented:

```
ORMenu toplevel_m "Help System";
ORMenu -> "About this game"
  with text "In this game, you play a...^^^...Feel free to send
             any bug reports or thoughts on the game to:
             ^^Jim@@64OnyxRing.com^^I hope you enjoy the
             game.^^^-Jim Fisher";
ORMenu -> "Author's Disclaimer" with text "I've
             never...^^^...It's only a story.";
ORMenu -> "Hints";
ORMenu ->-> "Avoiding the White Demon"
  with text[; print (italics)"~The white demon keeps coming
              into my castle and grabbing me!  How do I avoid him?~",
              "^^You ...^^^...best spent examining the world around
              you.";];
ORMenu ->-> "Who is this Bauefred, guy?"
              with text [; print (italics)"~This Baufred fellow
              keeps...","^^...later in the game.";];
ORMenu -> "About NPCs" with text "Blah, blah, blah....
              Put NPC Text here.";
ORMenu -> "Special Commands" with text "There are various
             commands that are atypical to other IF games, or at least
             are infrequent enough to warrant mentioning.  These can be
             divided into two groups:";
ORMenu ->-> "Commands specific to this game";
ORMenu ->->-> "Mamberflop"
  with text[; print "Mamberflop is a spell which...";]
;
ORMenu ->->-> "Ducalolly" with text[; print "The
             Ducalolly spell can...";]
;
ORMenu ->-> "Commands derived from the ORLibrary"
  with text "The ORLibrary offers several modules which add new verbs
             for the player. "
;
ORMenu ->->-> "Name"
  with text "The player has the ability to ~name~ an object and then
             refer to it by that name with a command such as:^^
             >NAME ORDERLY ~BOB~ ^^or^^ >REFER TO THE KNIFE AS A SWORD"
;
ORMenu ->->-> "Talk"
  with text "A generic addition to the ASK/Tell/Paradigm, the player has
             the ability to ~talk~ to another NPC...";
ORMenu -> "About the ~ORLibrary~"
  with text "The ORLibrary, or the ~OnyxRing~ Library is a framework of
             extensions to the standard library.  It contains...^^
             ...in finding out more about the ORLibrary framework,
             point your browser to: www.OnyxRing.com."
;
ORMenu -> "About the Author"
  with text "Jim Fisher, a software engineer by profession for close
             to a decade, has programmed in a dozen or more languages
             including Assembler, C/C++, C#, Delphi, Java, LISP, Perl,
             and SQL.  He has..."
;
```

Calling the menu programmatically can be accomplished with a call to the global routine `DoORMenu()`:

```
DoORMenu(toplevel_m);
```

Note that if a menu object is NOT specified to `DoORMenu`, then the routine checks the variable `ORMenuContext` for a menu object. Changing this variable based upon game state allows context sensitive menus to be implemented. If this variable does not define an ORMenu instance then the routine attempts to "guess" and find a top level menu object on its own.

ORActionQueue

ORActionQueue implements a queue to store player actions. Unless interrupted or made impossible, these actions will execute sequentially without pause. There are a variety of ways to use ORActionQueue. For example, it might be used to create an alternative to the standard library's implicit actions, which normally occur out-of-time. It could also be used with the ORPathMaker module to implement a command such as GO TO THE LIBRARY in which the path to the library is determined and the appropriate GO commands are generated.

Adding Actions to the ActionQueue

The `ActionQueue` object itself provides the `AddAction` routine. Use this routine to push onto the queue instances of the `ORQueuedAction` class (a derivative of `ORAction`).

Wait 3

From time to time, someone posts a question to RAIF about implementing a WAIT <number> command. As the following code demonstrates, ORActionQueue provides one mechanism for implementing such a verb:

```
extend "wait" first
   * number 'turns' -> WaitNum
   * number -> WaitNum
;
[WaitNumSub t;
   if(noun>ACTIONQUEUE_BUFFER) {
       noun=ACTIONQUEUE_BUFFER; print "[Maximum number of
         queued actions is ",ACTIONQUEUE_BUFFER,".]^^";
   }
   for(t=0:t<noun-1:t++)  !--queue up all but the first one
       ActionQueue.AddAction(ORQueuedAction.create(##Wait));
       <<WAIT>>; !--go ahead and do the first one now
];
```

The following transcript illustrates (note that only the emboldened text is actually typed by the player):

```
Room
Just a room, drab and bare.

>wait 3
Time passes.

>Wait

Time passes.
```

```
>Wait

Time passes.
```

The `AddAction` routine's optional second parameter allows you to override the text that is displayed when the queued action is performed. In the previous example, we could have supplied it:

```
ActionQueue.AddAction(ORQueuedAction.create(##Wait),
     "Meditate on spiritual matters");
```

Resulting in a similar transcript:

```
Room
Just a room, drab and bare.

>wait 3
Time passes.

>Meditate on spiritual matters

Time passes.

>Meditate on spiritual matters

Time passes.
```

Although it is not necessary for the display text to actually be parse-able (the above example would require additional programming to make it so), it is usually a good idea, since players will often look at the generated text and try it on their own.

Buffer Size

The number of actions that can be queued up at one time is determined by the ACTIONQUEUE_BUFFER constant which is defaulted to five. You can define this before the inclusion of VERBLIB to change the default. Notice that, in the WaitNumSub example above, the number specified at input was validated against ACTIONQUEUE_BUFFER.

Interrupting Queued Actions

Execution of queued actions will cease when the player has won or died, the ActionQueue object's interrupt property returns true, or an impossible action is encountered.

The first two scenario's, player death and manual intervention via the interrupt property, are self-explanatory and are accomplished respectively with:

```
deadflag=1;
```

or

```
ActionQueue.interrupt=true;
```

The third scenario, impossible actions, is primarily useful in games where the world state is subject to change. For example, supposed there exists in the game world a locked door, the key to which resides in a nearby closed box. The following lines should queue up the necessary actions needed to open the door:

```
ActionQueue.AddAction(ORQueuedAction.create(##Open, box));
ActionQueue.AddAction(ORQueuedAction.create(##Take, key));
ActionQueue.AddAction(ORQueuedAction.create(##Unlock, door, key));
ActionQueue.AddAction(ORQueuedAction.create(##Open, door));
```

But suppose an NPC has previously removed the key and wandered away with it. When the player opens the box, it is empty and all remaining queued actions are impossible. When such a situation is encountered, the action queue is cleared and the player is notified:

```
[Queued actions are no longer possible.]
```

ORNameable

To create a nameable object, simply derive it from the `ORNameable` class:

```
object golem "Golem"
  has   animate
  class ORNameable
  with description "The creature looks dirty and pathetic."
  ,   name 'golem'
;
```

During game play you can name the golem with a command similar to:

```
>name golem "bob"
```

After this you can begin referring to the golem by name:

```
>x bob
The creature looks dirty and pathetic.
```

ORPathMaker

The ORPathMaker object will determine a path between two rooms. Two starting objects are taken by the `determine_path` routine. These can be rooms, characters, or generic objects. Paths through doors is resolved; however, if a door requires a keys the starting object (the first parameter) must hold the needed key. This allows an NPC with the appropriate key to successfully take a path that an NPC without the appropriate key could not.

Sample use:

```
ORPathMaker.determine_path(player,bathroom);
ORPathMaker.determine_path(bathroom,bedroom);
```

The return values for this routine are:

-1 :	There is no connecting path between the two given objects.
-2 :	Ran out of workspace while trying to calculate path. Define the constant PATHDEPTH with a larger than default value.
-3 :	Although there WAS a path correctly determined, there was NOT enough space in the path property to contain it.
room obj :	The first connected room that must be traveled to in order to finally reach the destination.

Once the path has been correctly determined, the `size` property contains the number of steps in the determined path; the `path` property contains the final path itself.

There is a limited amount of scratch workspace used to determine a correct path. If there is not enough space, the path will not be findable. The constant `PATHDEPTH` is used to allocate this scratch space. By default it is set to 30, however it can be defined at a greater value in the main program's source.

ORPrefixSuffix

Creating an object with a period in the name -- like "Mrs. Robinson" -- introduces complications for the player. For instance...

```
>EXAMINE MRS. ROBINSON
```

...will cause the parser to stop parsing the input at the period and treat it as two separate commands:

```
>EXAMINE MRS
```

and

```
>ROBINSON
```

Thus, assuming there is only one object which the word "Mrs" can refer to, you will get the description followed by the "That's not a verb I recognize" error message. The parser fails entirely if the prefix can refer to multiple objects (e.g.: when "Mrs. Robinson" stands next to "Mrs. Baker").

This module scans the input and removes periods that follow general prefixes, such as "Mr" "Mrs" "Dr" "Col", etc... Keep in mind that periods are illegal in dictionary words.

ORReverseDirection

This is a short, simple routine, not at all worthy of its own module but used by multiple library entries and so is placed in its own module for the sake of efficiency.

ORReverseDirection simply takes a direction property or a direction object as a parameter (such as `nw_to`) and returns the reverse direction (in this case, `se_to`). It is used by several ORLibrary modules, such as ORNPCVerb, ORNPC_movement, and ORDynaMap.

For clarity, it should be reiterated that the ORReverseDirection routine also resolves direction objects such that passing in `s_obj` as a parameter will yield `n_obj` as a result.

ORSpecializedExit

This module extends the standard behavior for `exit` so as to allow a noun. A peculiarity of the standard library is that it simply doesn't like a noun specified with the `exit` verb. It is natural for a player who has just issued the command: GET ON TABLE or try to: GET OFF TABLE. Strangely, without modification, the standard library only supports EXIT, or GET OFF. With this module, commands such as...

```
>EXIT BALLOON
```

or

```
>GET OFF TABLE
```

or

```
>GET DOWN FROM THE CABINET
```

...or many other variations will work.

ORAdjective

The DM4 Exercise 75 gives an example of a ParseNoun routine which distinguishes between nouns and adjectives. ORAdjective implements a modified version of that routine, but leverages the OREntryPoints module so that other ParseNoun-like routines can co-exist without conflict. Additionally, this version of ParseNoun will also work with the ORRecogName module.

The usefulness of adjectives is apparent when dealing with similar objects. Consider the following three items:

```
object -> glass "glass" with name 'glass';
object -> marble "glass marble" with name 'glass' 'marble';
object -> marble_table "marble table" with name 'marble' 'table';
```

Notice how the parser is unable to distinguish the player's obvious intentions when referencing the marble:

```
You can see a glass, a glass marble and a marble table here.

>get marble
Which do you mean, the glass marble or the marble table?
```

In order to distinguish between the marble and the table, the adjective must be specified (i.e.: "glass marble" or "marble table"). This scenerio becomes even more complicated when trying to refer to the glass. Since it has no additional adjectives, it cannot be referenced at all:

```
>get glass
Which do you mean, the glass or the glass marble?
```

In order to make the glass uniquely referable, the designer must manufacture another word (perhaps 'drinking').

The introduction of adjectives when including the ORAdjectives module, is accomplished through the `adjective` property. The following code snip demonstrates:

```
object -> glass "glass" with name 'glass';
object -> marble "glass marble"
   with name 'marble'
   ,   adjective 'glass' 'red'
;
object -> marble_table "marble table"
   with name 'table'
   ,   adjective 'marble'
;
```

Adjectives will now be used to distinguish between similar objects (say, a wooden marble and a glass marble) and will no longer confuse objects with similar adjectives:

```
>get marble
Taken.

>get glass
Taken.
```

Although the parser grants a higher preference to nouns than adjectives, objects can still be referred to by adjectives alone:

```
>get red
Taken.
```

Some authors may wish to exclude adjective-only references, imitating the DM4's Exercise 75. This can be achieved by adding the following line to the `initialize()` routine:

```
ORAdjective.mode=REQUIRE_NOUN;
```

Unlike the default mode of `PREFER_NOUN`, adjectives will now only be used for disambiguation and cannot be used alone:

```
>get red
You cannot see any such thing.

>get red marble
Taken.
```

ORMagic

Of particular use in stories of the fantasy genre, the ORMagic module implements a system for casting magic spells. The most frequently used features of ORMagic will be covered in this section; however there are additional features, such as "persistent" spells that continue their effect long after their initial casting, which we will avoid detailing in favor of simplicity.

1. The SPELLS and CAST Verbs

By including the ORMagic module -- that is, by defining the USE_ORMagic constant -- the SPELLS verb and the CAST verb are immediately available. Of course, not having defined any spells makes them fairly useless, but they are there regardless:

```
Cave
This is a small, empty cave with no exits. You have no idea
how you got in here in the first place.

You can see a rock here.

>SPELLS
You have no magic at your disposal.

>CAST MURM
I am uncertain what you want to cast.
```

2. Creating a Spell

The actual creation of a spell is a relatively painless experience. Below is an example spell called "MURM" which does little more than assign an attribute:

```
attribute murmuring;
ORMagic Murm "Murm"
  with name 'murm'
  ,    knownby selfobj
  ,    cast[;
          if(second==0)
               "The MURM spell must be cast upon something.";
          if(second has murmuring)
               print_ret (The)second," is already murmuring.";
          give second murmuring;
          print_ret "You cast the spell upon ",(the)second,
               ". It begins to spew forth a continuous
               muffled murmuring noise.";
];
```

There are two properties of particular noteworthiness. The first is the `knownby`property

which lists all characters that know and are able to cast the spell -- in this case, only the player. The second is the `cast` routine which is called when a character uses the CAST verb. Notice that it is the responsibility of the spell itself to determine if casting is possible and to ensure that the object of the spell has been specified. See how the presence of the spell changes the previous transcript:

```
>SPELLS
You have the following spells at your disposal:
    Murm

>CAST MURM
The MURM spell must be cast upon something.

>CAST MURM AT ROCK
You cast the spell upon the rock. It begins to spew
forth a continuous muffled murmuring noise.
```

3. Additional Notes

See that our SPELLS verb listed the spell by its name. A description of what the spell does will also be displayed if it is defined in the generic `text`property:

```
ORMagic Murm "Murm"
  with name 'murm'
  ,  text "Cause objects to murmur."
  ...
```

This makes the result of the SPELLS command a little more informative:

```
>SPELLS
You have the following spells at your disposal:
   Murm (Cause objects to murmur.)
```

As a final note, remember that the CAST verb is really optional. The name of our newly crafted spell can be used as though it were a verb:

```
>MURM ROCK
The rock is already murmuring.
```

ORWAE_Formatting

The standard library's `WriteAfterEntry` routine interrogates objects and prints qualifying text such as "(providing light)" or "(closed, empty and providing light)". The `WriteAfterEntry` routine is anything but straightforward and trying to fathom the rules that govern these snippets of qualifying text can be a tedious task. What follows is a brief and simplistic overview:

Without modification, the standard library uses a total of 17 library messages to qualify various combinations of six object states. Three of these object states are described only when an object is part of the player's inventory, and not when it is depicted in a room description. The other three are described in either case. The following table details these:

Message	When Is It Displayed?
worn	Inventory only
open	Inventory only
locked	Inventory only
providing light	Inventory or Room description
closed	Inventory or Room description
empty	Inventory or Room description

Since the messages are implemented individually, it takes seventeen to cover the possible combinations, even given that some are mutually exclusive. Adding additional states is cumbersome and often means adding four or five additional messages, not to mention hacking the library to test for the appropriate settings.

This module implements an extensible framework to ease the burden of defining new qualifiers as well as redefining or removing the existing qualifiers.

1. ORWAE_FullFormatting and ORWAE_PartFormatting

Two routine lists (see `ORRoutineList`) have been created to manage the inventory and room description qualifiers. The routines registered with these lists determine if an object meets the required criteria and print the appropriate text if so. The following example creates two new qualifiers. One (sticky) will only be listed when part of the player's inventory while the other (muddy) will be described regardless:

```
attribute sticky;
attribute muddy;

[Initialise;
    ...
    !--muddy
    ORWAE_FullFormatting.add_routine(wae_muddy); !--in inventory
    ORWAE_PartFormatting.add_routine(wae_muddy); !--in room descrip
    ORWAE_FullFormatting.add_routine(wae_sticky);!--in inventory
];
[wae_muddy obj suppress;
    if(obj has muddy && suppress==false) print "covered in mud";
    if(obj has muddy ) rtrue; rfalse;
];
[wae_sticky obj suppress;
    if(obj has sticky && suppress==false) print "sticky";
    if(obj has sticky) rtrue; rfalse;
];
```

To demonstrate, let's create a couple of objects that have attributes which will be described in parentheses:

```
object box "box" cave
  has openable ~open container lockable locked muddy
  with description "Just a box."
  ,   name 'box'
;
object gumball "gumball" cave
  has sticky
  with description "A gumball, recently chewed."
  ,   name 'gumball' 'gum' 'ball'
;
```

Note in the following transcript, how our new qualifiers blend seamlessly with the traditional ones:

```
Cave
This is a small, empty cave with no exits. You have no idea
how you got in here in the first place.

You can see a box (closed and covered in mud) and a gumball here.

>TAKE ALL
box: Taken.
gumball: Taken.

>I
You are carrying:
  a gumball (which is sticky)
  a box (closed, locked and covered in mud)
```

2. Routine specifics

The qualifier-printing routines that we created above (`wae_sticky` and `wae_muddy`) take two parameters. The first is the object to be tested; the second is a flag which

determines whether or not the text should be suppressed. In either case, the routine should always return `true` or `false` to indicate that the object either does or does not meet the necessary criteria for printing the text.

Six of these qualifier-printing routines have been created and registered with the appropriate routine lists to handle the traditional library-defined qualifiers. These are listed below.

_WAE_worn
_WAE_opencontainer
_WAE_lockedcontainer
_WAE_light
_WAE_closedcontainer
_WAE_emptycontainer

Remember that the first three have only been registered with the `ORWAE_FullFormatting` object by default. To remove these, use the `ORRoutineList` object's `remove_routine` method. For instance, the "(which is empty)" qualifier can be eliminated from a game entirely by adding the following lines to the initialise routine:

```
ORWAE_FullFormatting.remove_routine(_wae_emptycontainer);
ORWAE_PartFormatting.remove_routine(_wae_emptycontainer);
```

3. Freed Messages

Recall that the standard library defines 17 library messages to handle the possible combinations of the six different qualifiers. Since the combination of these qualifiers is handled programmatically within ORWAE_Formatting, messages ListMiscellany #7-17 are no longer used and are free to be redefined.

ORNumberedContainer

This is an object that is designed to emulate a collection of openable, stationary containers (like lockers).

The following properties are of particular interest:

- singular_name the name of a singular compartment (e.g. "locker").
- start_numb the minimum number assigned to the compartments.
- end_numb the maximum number assigned to the compartments.
- numbered_description this routine will print pertinent information about which compartments are open and what contents are available. It is useful to call this routine after the description.
- contained_obj & contained_in - these property lists hold the contents of each locker. There should be enough elements in these arrays to hold all potential contents. Both of these arrays should be the same size as there is a one to one relationship between the items in these lists.
- open_state - a list of the compartments that are currently open. If more compartments are provide than there is room in this list, then the potential exists for the open to fail. In this case the property "cannot_open_more_msg" is run which outputs an appropriate message.
- cannot_open_more_msg output when the player has tried to open more compartments than there is room for.
- add_item_to call to add an object to a specific container.
- remove_item call to remove an object.

As a note, the display name, or short_name as may be the case, is displayed when referencing the object as a whole and should be plural in form (e.g. "lockers"). The singular_name property is used when referring to a single construct so should be singular in form.
Also, although it is fine to redefine the array properties of this object, it might prove to be easier to define an array table and assign it upon initialization.

For instance:

```
array lockerstate table 1000;
```

```
[Initialise;
  numcont.open_state=lockerstate; !----allow 1000 lockers
                                  !---to be open at once.
];
```

The following example demonstrates using the ORNumberedContainer object:

```
MyObject tilecubby "tile cubbies" hall
  class ORNumberedContainer
  with name 'cubby' 'door' 'doors'
  , singular_name "cubby"
  , end_numb 9999
  , description[; print "Almost invisible, the tiles along the southern
              wall are actually tiny doors to little cubby
              spaces with numbers etched into them. There are
              thousands. ";
         self.numbered_description();
  ]
;
```

One way of loading an object into a locker is with the `add_item_to` routine. We'll do this in `Initialise()`:

```
MyObject gun "revolver"
  with name 'gun'
  , description "The gun is sleek and heavy, powerful and deadly. "
;
tilecubby.add_item_to(random(tilecubby.end_numb));
```

Part D
NPC Oriented Modules

Introduction to ORLibrary NPCs

Non-player characters (NPCs) represent, potentially, the most complex objects in a work of interactive fiction. Since they typically represent other people, NPCs should, in theory, be able to do all of the things that the player character can do. They might be able to hold a conversation with another character. They might be able to perform a chain of actions such as finding a key and then unlocking a door. Massive amounts of code have been written to shape NPCs in this fashion.

Inform's standard library supports NPCs to a limited degree. It defines the `animate` attribute to indicate something is alive, and leverages this attribute to modify the behavior of player commands. In this way the player can try to TALK to animate NPCs, but not a book. Likewise, trying to TAKE an NPC will generate an appropriate error message. Support for NPCs is further supplied by the `life` and `orders` properties which provide hooks for the developer to code reactions to player commands.

Conspicuously missing from the standard library's support of NPCs is the ability to do anything. Creating an NPC that can initiate an action requires designing a daemon. Actions are also strangely player-oriented. For example, the TAKE command allows only the player to take things. For an NPC to pick up an item, the designer must actually code a routine that moves an item into the NPC's inventory. This is complicated by determining scope for the NPC and taking into account the possibility of the item having the `static` or `scenery` attributes. Perhaps the item is within a closed, transparent case so that it is visible, but untouchable. What if an object's before rule prevents the TAKE action from succeeding? Even were all of these checks in place for NPCs, the standard library's default messages for actions are all coded to assume the player character is performing the action.

The ORLibrary's support for NPCs addresses all of these inefficiencies in the standard library. All standard action routines have been revised to support NPCs and a method has been put into place to cause an NPC to perform actions, even new verbs designed by the developer. The default messages have been revised as well. Additionally, the ORLibrary's NPC support provides a framework which can be leveraged to support goal-oriented NPCs that act of their own accord.

Conversations between NPCs and the player character as well as conversations just between NPCs are supported. Enhancements to the Ask/Tell model of conversation and an implementation of conversation menus is also in the

ORLibrary. Entire topic webs can be created where a conversation with an NPC flows naturally from one subject to another and branches in various directions.

Because not every NPC needs to do all of these things, NPC abilities have been packaged as opt-in components. This keeps a small footprint for simple NPCs and allows more complex NPCs to be granted new abilities with a single addition to their `class` property. The complexity of an NPC can therefore be limited and more easily managed.

What follows is a guide covering the various ORLibrary modules used create powerful NPCs. This includes the ORNPC component modules as well as the ORKnowledgeTopic modules. It should be noted that the internals of the NPC and knowledge modules are extremely complex and these articles will not detail their every nuance. They will, however, cover the most-used features and give examples geared towards getting developers up and running quickly.

ORNPC

As was mentioned previously, the standard library implements only minimal NPC support. Nevertheless, this support is the required foundation upon which more significant NPCs can be based. ORNPC builds upon this foundation to provide a more versatile and sophisticated base from which to construct NPCs.

Below is an example butler derived from the ORNPC class:

```
object -> butler "butler"
   class ORNPC
   with name 'man'
   , description "The butler ignores you, preoccupied with other things."
;
```

A little experimentation will show that the butler acts no differently than a bare-bones object with the standard library's `animate` attribute. So what is the use of the ORNPC class? The power behind ORNPC is not what it makes NPCs do, but rather the hooks it provides to enable NPCs to act on their own...

1. ORNPCControl

The core of NPC activity is powered by the `ORNPCControl` object. This object wraps the daemon which imbues life into an NPC. Each turn, the daemon calls an additive property routine called `heartbeat` which is defined in every ORNPC object. After `heartbeat` returns, a last chance to do something is provided by calling the `heartbeat_post` property if it is defined. The order in which NPCs act is determined by a property called `priority`. Lower priority NPCs act first.

Because all ORNPC objects are raised to life in this fashion, by a single daemon, it is an easy matter to halt the actions of all NPCs at once. A property in the `ORNPCControl` object named `pause` can be set to `true` to accomplish this. Likewise, it can be set to `false` to restart all NPC activity. This daemon is started automatically, and the call to `StartDaemon()` never actually needs to be implemented by the developer. Since starting and stopping all NPCs is not commonly done, the `ORNPCControl` object is often not referenced at all; however, there are a few occasions, such as the middle of a verb command, that it is convient to tell whether an NPC is acting of its own accord or following an order. During the NPC action phase, the `ORNPCControl` property `npcs_acting` is set to `true`.

It should be noted that the daemon does more than simply call the `heartbeat` routine. It also takes some steps to ensure that the world is as it should be for the NPC, just as it is for the player. This includes steps such as making the NPC the current actor and calling a replaced version of `MoveFloatingObjects()`.

2. Heartbeat

`Heartbeat` is the entry point for all NPC activity. Leveraging this property gives us a great deal of versatility. Many authors have found that the addition of even very small statements to remind the player of our NPCs presence can have a profound effect and go a long way toward making it appear more lifelike. Dan Schmidt does this with the tool man character in "For a Change" and leaves small, lasting reminders of his presence. Emily Short, also does this in her game "Metamorphosis" despite the fact that the only NPC in the game serves little more purpose than scenery. Sentences such as "The toolman jingles in the breeze" or "The gondolier lifts his head and sighs softly" fire at arbitrary times and add to the illusion of life. We can accomplish this easily with the heartbeat property:

```
 object -> butler "butler"
    class ORNPC
    with name 'man'
    , description "The butler ignores you, preoccupied with other things."
    , heartbeat[;
      if(random(5)~=1) rfalse; !--only do this 1 in 5 turns (average)
      if(PlayerCanWitness()==false) rfalse;
      switch(random(6)){
           1: "The butler clears his throat.";
           2: "The butler gives a small sigh.";
           3: "The butler looks around as though searching for something.";
           4: "The butler scratches his chin absent mindedly.";
           5: "The butler hums a tuneless melody.";
           6: "The butler figits.";
      }
      rfalse;
   ]
;
```

The `heartbeat` routine, like `life` and `before`, is an additive property. Provided newer versions do not interrupt it (by returning `true`), the default implementation of `heartbeat` will select a registered action for the NPC, and perform it. We'll cover registered actions in a moment, but before we do lets talk briefly about a routine called above that we've not yet discussed, the CanPlayerWitness() routine.

3. CanPlayerWitness

Unlike the `each_turn` property, the `heartbeat` routine is fired for an NPC regardless of the player's current location. This enables NPCs to act in ways that meet their own agendas even with the player is no where to be seen. For this reason, it is necessary to verify that the player is indeed able to see or hear the NPC before text is output. The `CanPlayerWitness()` routine can be used for this. Additionally, another object can be passed in if it is not the actor that we are checking (such as a CB radio that the actor is speaking through).

In the above example, the `heartbeat` routine simply exits if the player cannot observe the actions of the NPC, but care should be taken to ensure that this routine only filters the text output and does not filter out changes the NPC may make to the game world. In short, the `CanPlayerWitness()` routine should determine if the player is notified when an NPC picks up an object, but not whether the NPC actually *does* pick up the object.

4. The DoNothing_msg Property

When the NPC has considered all potential actions, it is possible that there is nothing to do. At this point the `DoNothing_msg` property will be run or printed if it is defined.

5. Registered NPC Actions

As mentioned previously, the default implementation of the additive `heartbeat` property will search for "registered actions" to perform. For the purposes of this discussion, "actions" are essentially member properties that have been registered with a call to the `register_action()` routine.

An action property takes the single parameter `can_perform`. When `can_perform` is passed in as `true`, then the NPC is attempting to make a decision about what action to perform. At this point, the action property should check and make sure that the action is possible (such as making sure there is an empty chair nearby if the action is to sit down) and return `true` if it is. When the `can_perform` parameter is set to `false`, then the action is actually being attempted and the appropriate code should be run.

Since some actions take more than one turn to complete, ORNPC implements a property called `continued_action` which an action routine can set to ensure it will be called again next turn. When `continued_action` is set to point at an action property, the normal steps in determining which action to take are circumvented. The action

being referenced will continued to be called from the `heartbeat` routine until the `can_perform` test returns false.

Alone, the ORNPC class does not implement any actions, but this framework is the basis for many of the NPC component classes. The `ORNPC_movement` and `ORNPC_converse` modules are examples of NPC actions.

6. Component Class Order

Recall that more advanced NPCs can be assembled from the NPC component classes. As you would expect, using a component class means simply referencing it on the NPC's class line; but due to the nature of Inform's implementation of multiple inheritance, the required order in which these component classes appear is unintuitive. For example, ORNPC should always appear last in this list, while component classes are added to the front of the list. To complicate matters, several component classes require placement between two other components.

In order to simplify this, each NPC component has been assigned an NPC Component Priority value (NPComPri) which is listed in the header of each source file. Higher priority components should appear further left in the class list, while lower components should appear further to the right. The below list summarizes the values for common NPC components:

ORNPC	0
ORNPC_AskTellLearn	10
ORNPC_Converse	15
ORNPC_DoVerb	5
ORNPC_Interact	10
ORNPC_MapKnown	15
ORNPC_Moods	5
ORNPC_Movement	10

ORKnowledgeTopic

The ORKnowledgeTopic class is an implementation of a topic-based conversation system. It aids in separating information from conversation so that two NPCs could potentially present the same information in two separate ways. It implements a topic memory, so that knowledge can be learned and taught to other NPCs. It records who has said what so that NPCs can respond differently when told or asked about the same information multiple times. It can be used to create inanimate knowledge sources, such as books that can be consulted. It further implements and unifies the entire ASK/TELL/ANSWER/CONSULT paradigm.

Examples for all ORKnowledgeTopic functionality would be a very large and go beyond this guide's intent for a quick introduction. In the future, expect additional documentation to surface describing more customized implementations of ORKnowledgeTopic. For now, we'll scratch the surface and cover the most common uses for this class.

1. Basic ORKnowledgeTopic usage

To have a conversation, we need to have someone to talk to. Below, find three NPCs all derived from a base class which provides a central location to make changes:

```
class npc_c
   class ORNPC
;
npc_c dragon "dragon" with name 'dragon';
npc_c soldier "soldier" with name 'soldier';
npc_c halfling "halfling" with name 'halfling';
```

Now we can create a simple knowledge topic, also derived from a base class, that we can talk about:

```
class topic_c
   class ORKnowledgeTopic
;
topic_c hello_halfling_t "hello"
    with knownby selfobj
    ,    name 'hello'
    ,    TopicInformation "~Hello, halfling,~ you say.
                ^^The halfling nods his head shyly."
;
```

We have now completely implemented a basic knowledge topic which we can successfully tell to an NPC:

```
>TELL HALFLING HELLO
"Hello, halfling," you say.

The halfling nods his head shyly.
```

Note that it is the knowledge topics themselves that keep track of who or what "knows" them. In this example, the player can only say "hello" because he/she is listed in the object's `knownby` property list. It is possible for two or more characters to "know" the same topic.

Knowledge topics can also have parents which are knowledge topics. When this is the case, a topic is also known by the same characters that know its parent. This enables "knowledge groupings" and offers a more flexible way to manage knowledge bases.

2. Common Knowledge

In addition to grouping topics by parent, there is another alternative. All children of the `CommonKnowledge` object are automatically known by every character in the game. That is, being a child topic of the `CommonKnowledge` object is equivalent to having the player and every NPC in the game listed in the KnownBy list.

3. Context & IsInContext

The hello_halfling_t topic given in the previous example has a significant drawback. It is completely possible to speak this topic to the wrong character:

```
>TELL DRAGON HELLO
"Hello, halfling," you say.

The halfling nods his head shyly.
```

Obviously the text is inappropriate. One possible remedy to this behavior would be to turn the `TopicInformation` property into a routine and modify the text based upon who is being addressed, however this is not a feasible for most topics.

The `IsInContext` routine is used to determine whether or not a topic can be told to a

given character. The default implementation checks to see if the character in question is listed in the `context` property. If so, then the routine returns `true`. Additionally, if the `context` property list is empty, then it is assumed that the topic is in context for everyone and `true` is returned in this case as well. For most topics, this is all that is needed, but it is possible to override `IsInContext` to check other criteria, like gender for example. For our current example, the default implementation works fine:

```
topic_c hello_halfling_t "hello"
   with knownby selfobj
   ,    name 'hello'
   ,    TopicInformation "~Hello, halfling,~ you say.
                ^^The halfling nods his head shyly."
   ,    context halfling
;
```

The `context` property can also be used to resolve ambiguity. Consider the following:

```
topic_c hello_dragon_t "hello"
   with knownby selfobj
   ,    name 'hello'
   ,    TopicInformation "~Hello, Dragon,~ you say.
                ^^The blows smoke in your face in
                acknowledgment."
   ,    context dragon
;
topic_c hello_soldier_t "hello"
   with knownby selfobj
   ,    name 'hello'
   ,    TopicInformation "~Hello, Soldier,~ you say.
                ~Greetings!~ the soldier replies."
   ,    context soldier
;
```

Without limiting the context, the parser would ask for clarification and the developer would be forced to supply additional words in the name property for the player to specify.

4. Asking for Information

Although TELLing a character something is good functionality, ASKing about something is a far more useful thing to do in a story. The topic's `query` property, if provided at all, determines how a question would be phrased to solicit the topic's information.

Consider the following new topic known by the halfling:

```
topic_c "dragon"
  with knownby halfling
  , name 'dragon'
  , query "~So, do you think you could out run this
                dragon?~ you ask."
  , TopicInformation "The halfling eyes the dragon for a
                moment. ~My uncle was eaten by a dragon,~
                he says. ~Were the dragon to try and chase
                me, I would probably meet the same fate.~"
;
```

Which enables the following transcript snippet:

```
>ask halfling about dragon
"So, do you think you could out run this dragon?" you ask.

There was no reply.
```

As was stated previously, the lone ORNPC class does little beyond the minimal functionality of the standard library. The three example NPCs given above cannot do anything on their own. They can be talked to, but cannot really answer the questions to which they know the answers. This ability can be added to a character with the `ORNPC_AskTellLearn` component module.

Wary reader may notice the absence of the `context` property in the above example. To provide clarity as to why it is not needed here, it may be worth the effort to restate what is likely obvious: While the `knownby` property defines who *knows* the topic, the context property defines who the topic can be *told to*. Since we are ASKing about a topic that the halfling knows rather than TELLing something that the player character knows, limiting the context is not necessary. It *will be*, however, when the `ORNPC_Converse` component makes our NPCs wily enough to ask questions on their own.

5. ASK/TELL Enhancements

There are two enhancements to the standard library's ASK/TELL implementation:

The first is the implementation of two abbreviations for ASK and TELL. These are A and T respectively. First seen (by this author at least) in Emily Short's game "Galatea", these have been commented upon by the IF community with varying degrees of favor.

The second is a mechanism that attempts to determine with whom the player is intending to speak.

These two features make it possible to issue the command:

```
>a dragon
```

...and have it translate into:

```
>ask halfling about dragon
```

ORNPC_AskTellLearn

The `ORNPC_AskTellLearn` module is an NPC component class. That is, it is a class which encapsulates a specific behavior that can be used in conjunction with other classes to piece together an effective NPC. As discussed in previous sections, basic NPCs do not come with the ability to relate knowledge topics, even though they may "know" the information. Adding the `ORNPC_AskTellLearn` class to an NPC's class list rectifies this and makes speaking to the NPC a more natural experience. In particular, the NPC will now be able to answer questions when asked and respond when told something. This is what was termed the "Librarian" model of conversation in the theory article NPC Conversations: Ask/Tell Theory, found at the following address:

http://www.onyxring.com/variousarticles.aspx?article=74

As is the case with several of the NPC and KnowledgeTopic classes, `ORNPC_AskTellLearn` contains a substantial amount of functionality that will not be documented here. Instead, we will cover the most commonly used aspects of this NPC component. To begin, recall from the previous section that the halfling was unable to respond when asked about the dragon. Simply adding this component to the front of the `class` property list of the npc_c class will change this:

```
class npc_c
   class ORNPC_AskTellLearn ORNPC
;
```

In addition to the KnowledgeTopic-derived object provided in the previous section, we'll go ahead and provide another example for easier reference:

```
topic_c "dragon"
  with knownby soldier
  , query "~You think you're faster than this hungry dragon?~
                you ask."
  , name 'dragon'
  , TopicInformation "The soldier looks up at the dragon.
                ~No,~ he concedes. Then a broad smile breaks
                out over his face. ~But I don't have to be,~
                he says looking down at his companion. ~I only
                have to be faster than the halfling."
;
```

Observe the new verbal ability of our NPCs:

```
>ASK HALFLING ABOUT DRAGON
"So, do you think you could out run this dragon?" you ask.


The halfling eyes the dragon for a moment. "My uncle was
eaten by a dragon," he says. "Were the dragon to try and
chase me, I would probably meet the same fate."


>ASK SOLDIER ABOUT DRAGON
"You think you're faster than this hungry dragon?" you ask.


The soldier looks up at the dragon. "No," he concedes. Then
a broad smile breaks out over his face. "But I don't have
to be," he says looking down at his companion. "I only have
to be faster than the halfling."

>ASK DRAGON ABOUT HALFLING
"I'm afraid I don't know anything about that," the dragon
says.
```

Notice the default message printed when an NPC doesn't know how to answer. The additional property `IDunno_msg` can be defined to override this message for specific NPCs. The below code demonstrates:

```
npc_c "dragon"
   with name 'dragon'
   ,   IDunno_msg "~Geroumph?~ sounds the dragon quizzically."
;
```

ORNPCVerb

The ORNPCVerb module is not an NPC component class. Instead, it redefines the majority of the standard library's verbs and enables them to be performed by NPCs as well as the player. Like the OREnglish "language definition file", this module exhibits no apparent difference when included alone in the compilation process.

In truth, the ORNPCVerb module is part of a triad of modules that work together to bring the NPCs of the game world under the same umbrella of rules that the player is subjected to. To put the components of this simulation in a nutshell: ORNPCVerb makes the standard library's verbs NPC compatible, OREnglish makes the default messages NPC compatible, and ORNPC_doverb empowers NPCs with the ability to perform actions.

Since this module is best exemplified when used with the `ORNPC_doverb` class, further covering will continue in that section.

ORNPC_doverb

Like ORNPC_AskTellLearn, ORNPC_doverb is a component class. It leverages the ORNPCVerb module and can be used to give an NPC the ability to perform actions in much the same way as the player does.

In a traditional Inform program, NPC actions must be coded directly. In order for an NPC to pick up an object and eat it, code like the following might be called from the NPC's `orders` property in response to an EAT order:

```
!--example of traditional NPC code
[NPCEat npc obj;
  print (The)npc," eats ",(the)obj,".";
  remove obj;
];
```

The problem with this approach is that it doesn't enforce any of the rules that are enforced for the player character. A slightly better implementation would be:

```
!--example of traditional NPC code
[NPCEat npc obj;
  if(parent(obj)~=npc){
       if(obj has static or scenery) print_ret (The)obj " cannot be
                picked up.";
       print (The)npc" picks up ",(the)obj,".";
       move obj to npc;
  }
  if(obj hasn't edible) print_ret (The)obj " cannot be eaten.";
  print (The)npc," eats ",(the)obj,".";
  remove obj;
];
```

Despite the extended code, this still isn't a complete implementation. No `before` or `after` routines are called to enable, for example, the rules a magic elixir would run. Nor are the `react_before` or `react_after` routines run which may interfere with the actions completely.

ORNPC_DoVerb addresses these issues by allowing NPCs to perform any action using the same verb routines that the player character uses. When derived from this class, an NPC's actions are initiated by the `DoVerb` property routine. For example, the following single line could accomplish what the above code does and still honor any rules like `before` and `after`:

```
npc.DoVerb(##EAT,obj);
```

This module also adds support to NPCs for following orders. The `follow_orders` property can contain a list of verbs that the NPC will follow. Alternatively, setting the `will_follow_all_orders` property to `true` will turn this support on for every verb available to the player. When this option is in use, a list of actions can be defined in the `ignore_orders` property to suppress certain verbs. This behavior can be usurped issued in the traditional manner via the `orders` property.

Below is a new version of our base NPC class. Note the inclusion of `ORNPC_doverb` on the `class` line to include this component in the class hierarchy. Also notice that `will_follow_all_orders` is set and the EXAMINE verb has been listed as an exception to this rule:

```
Class npc_c
  class ORNPC_doverb ORNPC_asktelllearn ORNPC
  with will_follow_all_orders true
  , ignore_orders ##Examine
;
```

In addition, let's add a couple of objects to our game world that the NPCs can interact with:

```
object table "table" has static supporter with name 'table';
object steak "steak" table has edible with name 'steak';
```

Now we can start bossing people around and see how the game rules bind our NPCs just as they do the player:

```
>dragon, get steak
Taken.

>dragon, eat steak
The dragon eats the steak.

>dragon, get table
The table is fixed in place.

>dragon, get halfling

The dragon does not suppose the halfling would care for that.
```

ORNPC_GoalDriven

The ORNPC_GoalDriven module implements an NPC component class. This class is an "NPC Behavior", also called an "NPC Action", which enables NPCs to pursue goals, possibly taking steps to make those goals possible. An NPC, derived from ORNPC_GoalDriven, is given a goal using the deceptively simple method `NewGoal`.

NewGoal

The syntax of `NewGoal` is reminiscent of the NPC_DoVerb module's `DoVerb` routine. There are behavioral similarities as well, although `NewGoal` is functionally more complex. Where `DoVerb` causes an NPC to attempt something immediately, regardless of whether the act is actually possible, `NewGoal` registers an act to be attempted during an NPCs "turn" whenever it seems possible.

To demonstrate this, consider the following one-room example:

```
!% +include_path=\mine\if\compilers\Inform\library,ORLibrary
!% +language_name=OREnglish
!% $MAX_INCLUSION_DEPTH=10
Constant USE_ORNPC_GoalDriven;
#Include "OR_Library_Include"; #Include "Parser";
#Include "OR_Library_Include"; #Include "VerbLib"; #Include "OR_Library_Include";

object room "Room" has light
  with description "Just a room with no exits,
          other than the floating door.";

object bob "Bob" room has proper
  class ORNPC_GoalDriven ORNPC_DoVerb ORNPC
  with name 'bob';

object TheExit "door" room
  has door openable ~open scenery
  with description "Just a door."
  , name 'door'
  , after[;
      open : if(actor == player) {
                 deadflag = 2;
                 print "^You open the door and win!^";
             }else{
                 print "^", (The)actor, " opens the door, so
                    surprising you that your heart stops.^";
                 deadflag = 1;
             }
             rtrue;
    ];

[Initialise;
  location = room;
  bob.NewGoal(##Open, TheExit);
  return 2;
];

#Include "Grammar"; #Include "OR_Library_Include"; end;
```

The point of this example is to open the door, after which the game ends. Note how `NewGoal` is used to register this objective with the NPC "Bob". The player gets

one chance to open the door before Bob does so himself. Inevitably, the game ends in one turn:

```
Room
Just a room with no exits, other than the floating door.

You can see Bob here.

>z
Time passes.

Bob opens the door, so surprising you that your heart stops.


   *** you have died ***
```

So Bob can open doors. That's good, but just how smart is Bob, really? Say the door is locked and the key resting in plain view. How well does Bob fair then?

```
object box "box" room has openable ~open container transparent
  with description "Just a box."
  ,   name 'box';
object key "key" box
  with name 'key'
  ,   description "A key.";
object TheExit "door" room
  has door openable ~open scenery lockable locked
  with with_key key

  ...
```

Bob isn't all that bright at the moment, although he does get points for jumping in at the last moment to "win" the game. His heart's in the right place:

```
Room
Just a room with no exits, other than the floating door.

You can see Bob and a box (which is closed) (in which is a key) here.

>z
Time passes.

>z
Time passes.

>open box
You open the box.

>take key
Taken.

>unlock door
(with the key)
You unlock the door.

Bob opens the door, so surprising you that your heart stops.


   *** you have died ***
```

Intelligence

Although Bob recognizes that the door is locked and not able to be opened, he isn't quite smart enough to open the box, take the key, or unlock the door with it. One way to address this is to create a new goal for each of these steps, but that isn't really practical since it presumes much about the game-state. What we really want is Bob to figure out for himself what steps he needs to take.

To make our NPC a bit smarter, we'll give him a `generate_goal_steps` property that returns `true`:

```
object bob "Bob" room has proper
  class ORNPC_GoalDriven ORNPC_DoVerb ORNPC
  with name 'bob'
  , generate_goal_steps true
;
```

Bob's IQ has just gone up considerably:

```
Room
Just a room with no exits, other than the floating door.

You can see Bob and a box (which is closed) (in which is a key) here.

>z
Time passes.

Bob opens the box.

>z
Time passes.

Bob takes the key.

>z
Time passes.

Bob unlocks the door.

>z
Time passes.

Bob opens the door, so surprising you that your heart stops.


   *** you have died ***
```

Foreknowledge

As part of the game, you might expect Bob to have advanced knowledge of the key's location. He doesn't. If we remove the `transparent` attribute from the box object, Bob doesn't know what to do. Of course, that changes the moment he sees the key:

```
Room
Just a room with no exits, other than the floating door.

You can see Bob and a box (which is closed) here.

>z
Time passes.

>z
Time passes.

>open box
You open the box, revealing a key.

Bob takes the key.

>z
Time passes.

Bob unlocks the door.

>open door

You open the door and win!


   *** you have won ***
```

To give Bob more intimate knowledge of his surroundings, an awareness of things unseen, set his `foreknowledge` property to `true`:

```
object bob "Bob" room has proper
  class ORNPC_GoalDriven ORNPC_DoVerb ORNPC
  with name 'bob'
  , generate_goal_steps true
  , foreknowledge true
;
```

Now Bob knows where the key is, even though it isn't visible:

```
Room
Just a room with no exits, other than the floating door.

You can see Bob and a box (which is closed) here.

>z
Time passes.

Bob opens the box.
```

GOAL_BUFFER

An NPC uses a scratch buffer while working out what needs to be done to accomplish his agenda. In the previous example, the act of opening the door required three additional steps to be generated before the door was unlocked and the goal could actually be accomplished. More complex goals will require additional

steps. The default size for goal calculations is five, but this can be easily changed. To increase the size of the buffer, define GOAL_BUFFER prior to including parser.

ORNPC_Converse

A cousin to `ORNPC_AskTellLearn`, `ORNPC_Converse` gives an NPC the ability to initiate conversations.

1. The TALK verb

In addition to defining an NPC component class, ORNPC_Converse also defines the TALK verb. TALK is very similar to the TELL verb except that it requires no subject be specified. Instead, TALK selects a random topic which is passed off to the TELL command. Since the ability to TALK was created as a standard verb, it is also usable by the player as well as NPCs.

2. The TALK Verb Algorithm

We should note that some effort is made by the TALK verb to select an appropriate topic. For instance, only topics that the NPC knows about are considered. Additionally, the topic must have an `initiatable` property set to `true` and must be in context for the character being spoken to (see the section on ORKnowledgeTopic for more information about the `IsInContext` property). Finally, either the NPC must not have already talked to the player about the topic, or the topic must supply the `repeatable` property with a `true` value to indicate that this topic can be said multiple times.

This default topic selection behavior implements the "Rambling Idiot" form of conversing, described in the article NPC Conversations: Ask/Tell Theory at the following URL:

http://www.onyxring.com/variousarticles.aspx?article=74

The "Conversationalist" model is also supported by ORNPC_Converse and will be described in more detail in the section on ORKnowledgeWeb.

To demonstrate this module, we'll continue our example with the Halfling, soldier, and dragon by first modifying our `npc_c` base class to derived from `ORNPC_Converse` as well as `ORNPC_DoVerb` and then creating some sample topics for them to volunteer:

```
class npc_c class ORNPC_Converse ORNPC_DoVerb ORNPC;
```

```
class topic_c class ORKnowledgeTopic;
...

topic_c "size"
  with knownby Halfling
  , context dragon
  , initiatable true
  , TopicInformation "~It's just my opinion,~ says the Halfling to
              the dragon. ~But you don't look as though you would be
              satisfied with a Halfling. We have very little meat on us. Now a
              muscle-bound human on the other hand, I would think he would
              make quite a tasty snack.~"
;
topic_c "catching"
  with knownby Halfling
  , context dragon
  , initiatable true
  , TopicInformation "~You look to be the sort of dragon that likes to
              work for his prey,~ the Halfling says to the dragon. ~I would think
              catching a well-conditioned soldier would be much more satisfying
              than slow, skinny little Halfling like myself.~"
;
topic_c "Halfling meat"
  with knownby soldier
  , context dragon
  , initiatable true
  , TopicInformation "~Don't be fooled by our apparent size difference,~ the
              soldier tells the dragon. ~Halfling meat is renowned for its flavor.~"
;
```

Observe one possible way the dialog could play out:

```
>Z
Time passes.

"It's just my opinion, " says the Halfling to the dragon.
"But you don't look as though you would be satisfied with a
Halfling. We have very little meat on us. Now a muscle-bound human on
the other hand, I would think he would make quite a tasty snack."

"Don't be fooled by our apparent size difference," the soldier
tells the dragon. "Halfling meat is renowned for its flavor."

> Z
Time passes.

"You look to be the sort of dragon that likes to work for his
prey," the Halfling says to the dragon. "I would think catching
a well-conditioned soldier would be much more satisfying than slow, skinny
little Halfling like myself."
```

3. Prolonged Topics

Support is included with this module for topics that last more than one turn (e.g.: ORKnowledgeScript). When a knowledge topic is selected, it is placed in the `current_subject` property. When it has been discussed to completion, that is, when the `HasBeenSpokenOfBy` routine returns `true`, it is removed and the `Current_subject` property is set to zero. While `Current_subject` is not zero, the `ORNPC_converse` action takes precedence over all other potential NPC actions.

A point of clarification should be made here: Although this seems to be related to the previously mentioned `repeatable` property, it is not. If the last topic talked about returns from from the `HasBeenSpokenOfBy` property, it will continue to be talked about regardless of the `repeatable` property.

4. The Can_Converse Property

Although an NPC may derive from this class, setting the `can_converse` property to `false` effectively disables this action.

ORKnowledgeWeb

Derived from ORKnowledgeTopic, ORKnowledgeWeb conversation topics empower characters that have *topic pools* enabling them to converse in a manner that flows sensibly from topic to related topic. Additionally, using ORKnowledgeWeb grants NPCs the ability to inquire about topics they do not already know.

This is not an NPC component class; rather its use, when combined with that of OROptionList, subtly alters the behavior of the TALK verb (see ORNPC_Converse). It implements the "Conversationalist" form of conversing, as described in the article NPC Conversations: Ask/Tell Theory at the following URL:

http://www.onyxring.com/variousarticles.aspx?article=74

Characters that do not provide a topic pool will treat ORKnowledgeWeb objects identically to ORKnowledgeTopic objects.

1. Topic Pool - OROptionList

Conceptually, a topic pool is a list of topics that a character might choose to speak of. If a character provides such a list, ORKnowledgeWeb objects will be added and removed from it as a conversation progresses. This list and the ability to manipulate its elements are provided by the OROptionList class. Although OROptionList has several uses and is *not* an NPC class at all, characters (including the player character) can still inherit from it to gain topic pool functionality. Continuing on from our previous example, the minimal changes required to enable topic pools are shown below:

```
class npc_c class OROptionList ORNPC_Converse ORNPC_DoVerb ORNPC;
class topic_c class ORKnowledgeWeb;
```

2. The Enhanced TALK Verb Algorithm

As discussed under "ORNPC_Converse", the TALK verb usually selects topics based upon the value of the `initiatable` property. Although this behavior continues to exist, it is usurped by the topic pool methodology. The default selection method is

used only when no suitable topics are found in the character's topic pool.

There are a number of ORKnowledgeWeb properties that affect how topics are handled under the new behavior.

2.1 Adding topics to a pool with RelatedTopics

Consider the three example topics and the game transcript from the ORNPC_Converse section. The single topic known by the solder, "Halfling meat", seems well placed in the transcript following the halfling's "size" topic, but all of the topics have the `initiatable` property set to `true`. Why weren't alternative, less sensible transcripts produced? The truth is that coincidence was the largest factor in selection order; slightly different circumstances could have influenced the conversation in undesired ways:

```
>Z
Time passes.

"Don't be fooled by our apparent size difference," the soldier
tells the dragon. "Halfling meat is renowned for its flavor."

"You look to be the sort of dragon that likes to work for his
prey," the Halfling says to the dragon. "I would think catching
a well-conditioned soldier would be much more satisfying than slow,
skinny little Halfling like myself."

> Z
Time passes.

"It's just my opinion, " says the Halfling to the dragon.
"But you don't look as though you would be satisfied with a
Halfling. We have very little meat on us. Now a muscle-bound human on
the other hand, I would think he would make quite a tasty snack."
```

What we really want is to stop the soldier from refuting the Halfling's size argument until after it has been made. To do this, we must first keep the soldier from volunteering the topic by removing the `initiatable` property (it defaults to `false`). Next we want add the topic to the soldier's topic pool, when -- and only when -- the Halfling has made his size argument. This act of "tying" one topic to another is accomplished with the `RelatedTopics` property.

The following code snip redefines the two topics appropriately:

```
topic_c "size"
  with knownby Halfling
```

```
   , context dragon
   , initiatable true
   , TopicInformation "~It's just my opinion,~ says the Halfling to
              the dragon. ~But you don't look as though you would be
              satisfied with a Halfling. We have very little meat on us. Now a
              muscle-bound human on the other hand, I would think he would
              make quite a tasty snack.~"
   , RelatedTopics halfmeat_t
   , NonSpecificTarget true
 ;
 topic_c halfmeat_t "Halfling meat"
   with knownby soldier
   , context dragon
   , TopicInformation "~Don't be fooled by our apparent size difference,~ the
              soldier tells the dragon. ~Halfling meat is renowned for its flavor.~"
 ;
```

Several topics can be listed in the `RelatedTopics` property.

2.2 Eavesdropping NPCs

In a story with only two characters, a conversation could go on for some time just by leveraging the `RelatedTopics` property. Telling something to one character gives him topics to respond with. This was almost enough for our example.

Almost.

Our sample is complicated by the fact that the Halfling isn't actually speaking to the soldier. A quick look at the `context` property shows us that the topic is directed at the dragon. Under normal circumstances, the related topics will only be added to the pools of the two characters actually having the conversation. Of course, this is not what we want here.

Since the Halfling is voicing this topic in a way that it can be overheard, we want all nearby characters to benefit from the `RelatedTopics` definition. Topics meant to be heard by multiple characters can be defined by setting the `NonSpecificTarget` property to `true`. See that this was done in the previous example.

2.3 Stunning Topics

There are points in a conversation when it makes sense to "drain" a character's topic pool. This is especially true when important, pivotal information is related. For example, revelation of a newly discovered murder might eclipse other, more mundane lines of conversation. Following such a declaration, authors may choose to prevent NPCs from announcing their lunch preferences.

Setting the `ResetOptionList` property causes the topic pools of conversing characters to be cleared prior to introducing the topics listed in the `RelatedTopics` property, effectively eliminating all conversational threads except those related to the current topic.

As with the `RelatedTopics` property, setting `NonSpecificTarget` causes this behavior to affect nearby characters as well.

2.4 Response-only Topics

Some topics, despite being listed in the `RelatedTopics` property, are not meant to be volunteered. Topics that return a `true` value for the `Response` property will be avoided by characters that know them. In such a case, only characters that are not alreay privy to its knowledge can ask about such a topic.

ORNPC_moods

Typically, when we code an NPC to conditionally take actions, it is based upon world elements. We could, for example, cause a maid to pick up anything near her that is out of place. While this is a necessary technique for determining whether or not an NPC can or will take an action, there is an additional factor that influences people which has thus far not been addressed: their state of mind.

The maid might choose to not pick up an object if she is in a bad mood. The ORNPC_moods class does not directly affect the NPC's behaviors; rather it provides a "mood system" for other modules to utilize. For NPCs that derive from this class, a state of mind is maintained and mechanisms for adjusting the NPCs frame of mind are provided. Additionally, these NPCs can remember how they feel about someone or something so that slapping an NPC, running away, and returning later does not result in instant forgiveness.

1. Agitating and Cheering Up an NPC

This class adds two routines to an NPC that effect its mood. The first, `Agitate`, adds a little irritation to an NPC's mindset when called. The second, `CheerUp` makes an NPC a little less irritable and a little more disposed toward happiness.

Both of these routines take a parameter named `silent`. When passed as true, the adjustment to the NPC's attitude is performed without feedback. This is useful when more than one call is made (when an event is really irritating). If `silent` is not true, then the appropriate message contained in either the property `cheerup_msg` or the property `agitate_msg` is printed.

2. The Mental State of an NPC

An NPC's attitude can be evaluated with a call to the property routine `MentalState`. This routine will return one of five predefined constants:

```
NPCLivid
NPCAngry
NPCNormal
NPCHappy
NPCEcstatic
```

It is important to understand that these values are a form of "mindset notation," a simplified representation, of the NPC's current mood. Aggravating an NPC in a "normal" state of mind will not necessarily make him "angry," but aggravate him enough and he will eventually become angry and finally cross the threshold into lividness.

3. Changing the standard of mindset

The "mindset notation" discussed previously is defined in a property routine called `FrameOfMind`. It is this routine that translates a broad-ranged numerical value into one of the five notation constants. `FrameOfMind` can be overridden to redefine the ranges at which an NPC is considered "Happy", or "Angry". A troll, for example, might never get happy at all.

4. Recalling and Recording Impressions

As we interact with the people around us, we remember our past experiences of them. We harbor grudges and play favorites based upon our likes and dislikes of the people we know, however it is seldom that we like someone by a logical choice. More often than not, we remember how we felt when last we interacted with them. To the point, we generally like or dislike people based upon how they make us feel.

Two routines are provided by NPCs that inherit from the `ORNPC_moods` class to implement this behavior. The first, `Record_Impression` takes an object as a parameter and will associate the way the NPC currently feels with that object.

As time passes, the NPC's mood may change, but his last impressions of an object, that is, how he felt at the time the impressions were made, can always be retrieved with a call to the routine `Recall_Impression`.

Both of these routines deal with a snapshot of the NPC's mindset and the result of `Recall_Impression` can be passed through the `FrameOfMind` routine to transform it into mindset notation.

5. Effect by Recollections

Our impressions of things do more than determine our likes and dislikes. They also affect our current frame of mind. When coming across a person he dislikes, even if currently in a good mood, a person's frame of mind will likely be adversely affected.

The property routine `affect_from` takes an object as a parameter. This routine looks for the object in the NPC's likes and dislikes and makes adjustments to the NPC's mindset accordingly.

ORNPC_movement

The ORNPC_movement module implements what was previously referred to as an "NPC action." As the name describes, this module gives the NPC the ability to initiate acts related to movement. Movement can be defined and configured in a number of ways, but before we discuss these, let's create a few simple rooms to serve as a context for our examples, and place the player in one:

```
object foyer "Foyer" has light
  with description "This is a clean, simple entry hall.
           ^^Exits lead north, east, and northeast."
  ,   n_to diningroom
  ,   e_to study
  ,   ne_to atrium
;
object diningroom "Dining Room" has light
  with description "This is really more of a breakfast nook.
           ^^Exits lead south, east, and southeast."
  ,   s_to foyer
  ,   e_to kitchen
  ,   se_to atrium
;
object kitchen "Kitchen" has light
  with description "The place where food is prepared.
           ^^Exits lead south, west, and southwest."
  ,   s_to study
  ,   w_to diningroom
  ,   sw_to atrium
;
object study "Study" has light
  with description "A library without books.
           ^^Exits lead north, east, and northeast."
  ,   n_to kitchen
  ,   e_to foyer
  ,   ne_to atrium
;
object atrium "Atrium" has light
  with description "Light filters down from above.
           ^^Exits lead northeast, northwest, southeast, and southwest."
  ,   nw_to diningroom
  ,   ne_to kitchen
  ,   se_to foyer
  ,   sw_to study
;
[Initialise; location=foyer; ];
```

1. NPC Wandering

By default, when an NPC has made the decision to move, it will wander. That is, the NPC will walk around in any available direction from which he did not just come, and turn around to retrace his steps only when there are no other exits. Additionally, wandering NPCs will open closed doors as they pass through, possibly unlocking them if they hold the key. What follows is a definition of a character capable of moving around and a single NPC derived from it. Take note that the NPC has been set up to obey the player's orders by virtue of the `will_follow_all_orders property`. See the section on NPC_DoVerb for more information on this.

```
Class npc_c class ORNPC_movement ORNPC_doverb ORNPC
    with will_follow_all_orders true
;
npc_c maid "maid" foyer has female with name 'maid', can_move false;
```

2. The HALT and UNHALT Orders and the Can_Move Property

Notice the `can_move` property in the maid. An NPC's ability to wander around can be turned off and on programmatically by flipping the `can_move` property to `true` or `false`. As we'll see in the transcript of the next section, this can also be done with the HALT and UNHALT orders.

3. The FOLLOW Verb

When dealing with autonomous NPCs, there are times that the player may want to follow an NPC around. To avoid the tediousness of scanning through text to find out what direction the NPC went, the FOLLOW verb has been implemented. Note, that the FOLLOW verb only works on NPCs that have just left and are therefore in an adjacent location. The following transcript demonstrates:

```
Foyer
This is a clean, simple entry hall.

Exits lead north, east, and northeast.

You can see a maid here.

>MAID, UNHAULT
The maid resumes a more mobile posture.

The maid wanders away to the east.

>FOLLOW MAID

Study
A library without books.

Exits lead north, east, and northeast.

You can see a maid here.

The maid wanders away to the northeast.
```

4. The FOLLOW Order and the Start_Following Property Routine

In addition to wandering around, an NPC can be made to follow another object's movements. This could either be another NPC, or perhaps an object that the NPC carries. The `start_following` property routine is used to specify the followed focus in code, but NPCs can also be ordered to follow an object with the FOLLOW order.

Although using FOLLOW as an order is similar to using FOLLOW as a command, it differs in that the behavior is persistent. That is, the NPC will continue to follow that object in subsequent turns until the `stop_following` property routine has been called, or the STOP FOLLOWING order is issued. To exemplify this, let's add another NPC:

```
npc_c butler "butler" foyer with name 'butler';
```

Notice that the butler does not have the `can_move` property turned off. This means that he will immediately start wandering around unless we catch him with the FOLLOW order first:

```
Foyer
This is a clean, simple entry hall.

Exits lead north, east, and northeast.

You can see a maid and a butler here.

>BUTLER, FOLLOW ME
The butler looks at you and then nods agreeably.

>MAID, FOLLOW BUTLER

The maid looks at the butler and then nods agreeably.

>N
Dining Room
This is really more of a breakfast nook.

Exits lead south, east, and southeast.

The butler enters from the south (following you).

The maid enters from the south (following the butler).
```

5. Using the Tracker Property in Conjunction with the ORPathMaker Module

A point important enough to restate was specified above in section 3: FOLLOW is usually limited to adjacent rooms only. If, in the previous example, the butler were to be transported to a room not adjacent to the maid's location (in this case, the study) then the maid would not be able to FOLLOW him. Instead she would fall back to wandering mode until she stumbled across him again at random.

Including the ORPathMaker module offers another capability which can be turned on by setting the `tracker` property to `true`. This gives the NPC the ability to "track," what they are following as a bloodhound might do. Distance is limited only by the constraints of the `PATHDEPTH` constant see the description of the ORPathMaker module

for details).

Keep in mind that the ORPathMaker module is not an auto-dependency. It must be include for this functionality to work; setting the `tracker` property alone will accomplish nothing.

6. NPCs Following a Path

As an alternative to wandering, a `path` can be defined for an NPC to follow in the `path` property list. An element of this list can either be a direction object or a neighboring room. If a neighboring room is specified, then the direction in which the room lies is determined and an attempt is made by the NPC to walk that way.

There are occasions where an NPC's `path` can be made impossible to follow. Assuming the defined `path` is valid, these occasions generally occur when an NPC's location has been changed by an external means. For instance, transporting an NPC to another location would do this, perhaps making the NPC "lost". When an NPC is "lost", it will wander until it reaches a location that it "recognizes" (is in its `path`), and resume following its `path` from there. It is important to note, that an NPC's ability to gracefully recover when diverted from its `path` is stunted by the use of direction objects in the `path` list rather than room objects. The NPC's ability to recognize its location is limited to the rooms in the `path` list. Perhaps the most versatile method is to use a combination of both.

By default, when the end of the defined `path` is reached, it is looped back to the beginning and started over. It is possible, however, to have an NPC retrace his steps and follow the `path` in reverse. This behavior can be implemented by setting the `reverse_at_path_end` property to true.

The following code demonstrates the same path set up for the butler and the maid, one using direction objects, the other using rooms. Note that the maid and butler will walk together for the first four turns, but because of the `reverse_at_path_end` property, the butler will then turn around and walk back the way he came.

```
npc_c butler "butler" foyer
   with name 'butler'
   ,   path n_obj e_obj s_obj w_obj
   ,   reverse_at_path_end true
;
npc_c maid "maid" foyer
   has female
   with name 'maid'
   ,   path diningroom kitchen study foyer
```

;

ORNPC_Map_Known

This NPC componant class modifies the behavior of `ORNPC_movement` so it is required that NPCs which derive from `ORNPC_Map_Known` also derive from `ORNPC_movement`. Remember that ORNPC_Map_Known should appear *after* ORNPC_movement on the class line (refer to the the section entitled "Introduction to ORLibrary NPCs").

One of the features of the `ORNPC_movement` class allows NPCs to follow a path. There are restrictions to this, such as the rooms in the path need to be contiguous. If the NPC ever arrives in a room which is not adjacent to the next location listed in his path, then the NPC can become "lost" and wander around until he stumbles across a location that he "recognizes".

This module removes that restriction and allows rooms in the NPC's `path` property to be separated by several other rooms. It leverages the `ORPathMaker` module to generate the shortest path between the NPC's current location and target location thereby enabling the NPC to arrive at his path and never wander around lost (assuming the destination is actually reachable and falls within the limitations of the `ORPathMaker` object.)

Other than adding the `ORNPC_MapKnown` class to the NPC's inheritence chain (and prerequisits) no additional code need be implemented to make this functionality occur.

Part E
Other Stuff

Using the ORLibrary Entries Without the Framework

The modules in the ORLibrary were written to take advantage of the ORLibrary framework, specifically the OR_Library_Include.h file. As a rule, it is **substantially** less trouble to simply utilize this framework, but for those who are adamant about using a module alone, it is possible, although sometimes problematic, to use a library entry without committing yourself to the framework.

Including the Extension

As with traditional extensions, ORLibrary modules are imported using the `include` directive. An ORLibrary module differs from the standard extension in that it must be included four times, in four different places:

- `Before inclusion of Parser`
- `Before inclusion of VerbLib`
- `Immediately after the inclusion of VerbLib`
- `Immediately after the inclusion of Grammar`

ORLibrary modules attempt to detect discrepancies in the order of inclusion and will throw a compile-time error if one is detected.

Limitation by Dependencies

An increasing number of ORLibrary modules depend upon other modules and attempt to meet these dependencies automatically by using framework techniques. Modules that have dependencies will detect if the framework is present and throw an error if it is not.

By convention, a list of a module's dependencies can be found in the "AutoDep" section of its comment header. An example of such a module is ORKnowledgeTopic, which depends upon OROptionList, ORDynaString and ORPronoun.

Manually Meeting Dependencies

It is still possible to convince a module with dependencies to compile without the framework. Do this by defining the `ORLibraryInclude` constant at the start of your source file. This "fools" the module into believing the framework will be there to satisfy it's needs. Since it really isn't, the burden of meeting these requirements falls to the developer. Keep in mind that many dependencies have dependencies of

their own. So: when including an extension manually, all requisite modules, including requisites of requisites, must also be included manually.

The following code skeleton demonstrates leveraging the ORKnowledgeTopic extension without the library:

```
Constant ORLibraryInclude; !--Convince modules we are using the framework

!... code that belongs before PARSER
#include "OROptionList";     !--dependency of ORKnowledgeTopic
#include "ORTextFormatting"; !--dependency of ORDynaString
#include "ORPronoun";        !--dependency of ORKnowledgeTopic
#include "ORDynaString";     !--dependency of ORKnowledgeTopic
#include "ORKnowledgeTopic";
#include "parser";           !--standard library inclusion

!... code that belongs between PARSER and VERBLIB

#include "OROptionList";
#include "ORTextFormatting";
#include "ORPronoun";
#include "ORDynaString";
#include "ORKnowledgeTopic";
#include "verblib";          !--standard library inclusion
#include "OROptionList";
#include "ORTextFormatting";
#include "ORPronoun";
#include "ORDynaString";
#include "ORKnowledgeTopic";

!... code that belongs between VERBLIB and GRAMMAR

#include "grammar";          !--standard library inclusion
#include "OROptionList";
#include "ORTextFormatting";
#include "ORPronoun";
#include "ORDynaString";
#include "ORKnowledgeTopic";

!... code that belongs follows GRAMMAR
```

Part F
Quick Start

How do I quickly setup and use the ORLibrary?

I've been compiling Inform programs for a while, and just want to quickly use the ORLibrary without juggling directories. What's the easiest way?

The easiest way to **setup** the ORLibrary is simply to dump all the ORLibrary files into the same directory as the standard library and include the following text on your compiling command line:

```
+language_name=OREnglish
```

That's it. The ORLibrary is now setup and ready to be used.

The easiest way to **use** the ORLibrary is to leverage the OR_BlankGame.inf template as a starting point for your game. It already contains the required four includes at the appropriate locations and provides a list of commented-out references to most library entries. Simply uncomment the ones you would like to use and they will automatically be included.

How can I avoid specifying an object's name?

Modules used: ORRecogName.h
Auto Dependencies: None

I have a simple wooden cube that I would like to refer to. It doesn't make sense to me why I have to specify the "name" property when I've already named it. I just want to do the following:

```
object -> wcube1 "wooden cube";
```

How can I make this work?

The ORRecogName entry does this automatically. Simply include (or uncomment) the USE_ORRecogName constant in your source file and the cube can be referenced by any of the words printed by in it's textual name. The "name" property now needs only to be specified for synonyms and plural words (with the //p indicator).

"How can I generate the map from the player's moves?"

Modules: ORDynaMap
Auto Dependancies: None

I'm planning a scene in a game where the player needs to wander around for a while before reaching his destination. I don't want to create a large map because it could take the player too long to find the destination, but I also don't want him to "stumble" upon the end in just a single move.

I played a game* once where a map of the Martian landscape was generated by the player's choice in movements, giving the illusion that the map was bigger than it was. I wanted the player to wandering through about four rooms altogether. How easily can this be done?

This can be done very easily with the ORDynaMap module. Just follow three simple steps:

1) Add (or uncomment) the USE_ORDynaMap constant.

2) Create the "wandering rooms" which will be generated. It is easiest to derive these from the ORDynaMapRoom class:

```
ORDynaMapRoom forest_start "forest clearing"
  with description "The beginning of a vast forest."
  ,    s_to "Don't go that way. You just came from there.";

ORDynaMapRoom forest1 "forest"
  with description "You are surrounded by trees.";

ORDynaMapRoom forest2 "forest"
  with description "No place to go but toward the trees.";

ORDynaMapRoom forest3 "forest"
  with description "You are definitely lost in the woods.";

ORDynaMapRoom forest_end "forest clearing"
  with description "At last! You've found the forest, hidden
              by all the trees!";
```

3) Create an instance of the ORDynaMap class and specify the rooms in the "found_in" clause in the order that they will appear:

```
ORDynaMap found_in forest_start forest1
             forest2 forest3 forest_end;
```

That's it. obviously the player needs to have access to forest_start. From there, any direction not already defined will lead to the forest1 room and the map will be altered accordingly.

**The game was "Photopia" by Adam Cadre.*

How can I easily setup a door?

Modules used: ORDoor.h
Auto Dependencies: None

I'd like to implement several doors in my game, and have been looking at sample code. Yuck! Having to specify the door_dir, door_to, and found_in properties is a lot of complexity that I would like to avoid. What would really be nice is to just create a door and point to it from the room's direction property. Is this possible?

Sure. The easiest way to implement a door is to define (or uncomment) the USE_ORDOORINIT constant in your source file. The following code will now work:

```
!---Two rooms with a door between them...
object east_room "East Room" has light
  with description "The eastern room. A door lies to the west. "
  ,     w_to door1;
object west_room "West Room" has light
  with description "The western room. A door lies to the east. "
  ,     e_to door1;
!--- And now the door...
ORDoor door1 "door" with name 'door';
```

How can I make an NPC follow the PC?

Modules: ORNPC_movement

Auto Dependancies: ORNPC, ORNPCVerb, ORNPC_doverb

I've got an NPC that I want to make follow something else around. Is there an easy way to do this?

Yes. The ORNPC_movement class supports this functionality implicitly, using "NPC verbs". Simply define (or uncomment) the USE_ORNPC_Movement constant and all needed library entries will be pulled in.

At this point I'd like to mention that NPCs are especially complex and a considerable amount of work has been put in them. The actual ORLib documentation should be consulted for any real understanding of the ORNPC component classes, but for a quick answer to the question, the following NPC will follow the player character around:

```
object bob "Bob" east_room has proper
  class ORNPC_movement ORNPC_doverb ORNPC
  with name 'bob'
  , follow_object selfobj;
```

The reference to the "selfobj" object can be changed to any object in the game, including carriable items and obviously "Bob's" location is game dependant.

How do I say something to an NPC?

I've created a small game with a single NPC and a single room. I'd like to talk to the NPC. How can I accomplish this?

What you've described so far probably looks something like this:

```
#Include "OR_Library_Include";
#Include "Parser";
#Include "OR_Library_Include";
#Include "VerbLib";
#Include "OR_Library_Include";
object theroom "room"
  has light
  with description "This is just a room."
;
object bob "Bob" theroom
  has animate proper
  with name 'bob'
;
[Initialise; location = theroom; ];
#Include "Grammar";
#Include "OR_Library_Include";
end;
```

The above example is a pointless exercise as far as the ORLibrary is concerned. Since no modules are specified, the included file "OR_Library_Include" has no function. The resulting program runs identically to a plain vanilla Inform program.

The ability to talk to NPCs is provided by the ORKnowledgeTopic module. With very little work (only three additional lines of code after referencing the module) the player can begin to speak to other characters immediately. For clarity, the new lines have each been emboldened:

```
Constant USE_ORKnowledgeTopic;
#Include "OR_Library_Include";
#Include "Parser";
#Include "OR_Library_Include";
```

```
#Include "VerbLib";
#Include "OR_Library_Include";
object theroom "room"
  has light
  with description "This is just a room."
;
object bob "Bob" theroom
  has animate proper
  with name 'bob'
;
[Initialise; location = theroom; ];
ORKnowledgeTopic with name 'hello'
   , knownby selfobj
   , topicinformation "~Hello there,~ you say."
 ;
#Include "Grammar";
#Include "OR_Library_Include";
end;
```

The code of interest details the ORKnowledgeTopic instance. Other than the traditional "name" property, only two properties are really needed for this basic conversation topic to work:

The first is the "knownby" property which lists all the characters that know this topic and can therefore speak about it. Only the player object, that is "selfobj," knows this topic.

The "TopicInformation" property stores the actual text that is printed when the topic is spoken.

A short transcript shows that you can now talk to Bob:

```
room
This is just a room.
You can see Bob here.


>tell bob hello
"Hello there," you say.
```