Compiled from the OnyxRing article archive on 2/27/2004 1:13:54 AM

Table Of Contents

Part A
About This Guide

An Inform Developer's Guide: Introduction to the printed version

This guide, in its latest form exists at www.OnyxRing.com. All of the articles contained here are stored as HTML fragments in a database. When a browser requests an article the HTML fragment is pulled from the database, munged together with generic web-site code and displayed. This is how the site's article archive has been designed and it has operated successfully in this fashion for years now. One page, one article. For a time, everything was good... or so I thought.

As the web site began to accumulate more and more regular hits, and the number of articles began to grow, I began to receive innocent e-mail requests:

> *"Mr. Fisher, I really love your site, but I really would like to print off all the articles at one time. Can you produce a single version that I can print off?"*

> *"Hey OnyxRing guy! It hurts my eyes staring at the screen all the time. Can you produce a single version of your Inform Guide?"*

> *"Jim, I'm tired of this whole Internet thing and I really hate having to look at your site's articles one at a time. Can you produce a PDF file or something?"*

> *"Look you lazy, Dweeb! You should quit holding Roger Firth's awesome TextPad article hostage for site-hits and share it everyone else. Why don't you make a PDF file or something. The DM4 and IBG are both in that format. It can't be that stinking hard, you selfish twit!"*

So, deciding to give in to this constructive stream of suggestions, I began to look into what it would take to do a conversion. Obviously a major concern was the frequency in which the site articles changed. A conversion needed to be relatively painless since it would need to be updated frequently.

In the end, being a somewhat lazy individual (as one person pointed out), I wrote a little code to do this for me. My HTML to PDF converter took a few weeks to write, but seemed to be more or less successful at what I was trying to accomplish. If you are reading this from Adobe Acrobat or flipping through a printed hard copy, then

you are reviewing the result.

Some of the articles translated extremely well to the printed version. Others translated with a lesser degree of success. For example, the FAQ -- which was designed to be read online -- is riddled with references to Cascading Styling Sheets and the Article Rating System (neither of which exist in the printed version). Hyperlinks also have not been translated so particularly ugly remarks akin to "click *here* to see such-and-such" seem particularly out of place. As time goes by, I hope to modify the articles to make them more agreeable to both web and printed mediums.

Despite short comings in the translation, this product represents a sizeable amount of work, and I am proud to present what has been repeatedly requested of me for several years: An Inform Developer's Guide in PDF form, the most recent version of which can be found at:

www.OnyxRing.com/downloads/AIDG.PDF

Thank you,

Jim Fisher

An Inform Developer's Guide: Introduction

et me begin with some clarification. As the name states, this is not "THE Inform Developer's Guide." There are already several works on the net that qualify as developer guides for the Inform language. Hands down, the most widely viewed of these in the one written by the author of Inform himself, Graham Nelson. If any work deserves to be called "THE Inform Developer's Guide" it is that one. Instead, the name of this manual is "AN Inform Developer's Guide." Again, that's "an", as in "one of several."

For those who do not know the history, Graham Nelson contributed greatly to the IF community. This highly esteemed Oxford professor wrote the Inform compiler, the standard Inform Library, and several acclaimed games. He wrote a number of articles on the topic of adventure game design. Additionally, he wrote "The Inform Designer's Manual." Without a doubt, Mr. Nelson made a tremendous mark upon the IF community.

But Graham Nelson is only one man.

Different people learn in different ways. Some learn visually, while others learn audibly. Some need to see a concept applied to understand its use. Others need only the theory. Consider all the different ways to present a topic. Different methods work best for different people. No single man can cover the entire spectrum of presentation.

That's where this guide comes in. You see, I'M not writing it, YOU are. When a developer learns Inform, he/she usually does so because of an idea for a game. Every game is unique and each game applies aspects of IF that others do not. By the time the game developer has finished writing it, he/she has also acquired a bit of personal experience in design. It is this personal experience that you will find in this guide, for the articles here are written by developers like you. One developer may have a fair amount of experience developing NPCs. Another might have written a game with extensive language modifications. It is by submitting articles to this guide that other developers can share their experience with you.

It is by submitting articles to this guide that you can share your experience with other developers.

An Inform Developer's Guide FAQ

About the site:

Q: So is this a replacement for Graham Nelson's Designer's Manual?

A: Certainly not! The Designer's Manual is the undisputed standard. This guide should be considered a supplement. As described in the Introduction, it is the philosophy of this guide that different people learn in different ways. There may be some duplication of content between this guide and the Inform DM, but a developer with a different personal take will write it. It is our belief that the same information conveyed in different words, from a different perspective, will often help to make the topic clearer.

Q. What is this "rating" stuff?

A. Since the articles stored here are both "by" and "for" Inform developers, it makes since that those same Inform developers rate whether or not they felt the article was helpful. Article lists, such as the Table of Contents, show a ratio of positive votes over total votes in parentheses. Additionally, the total number of times the article has been viewed (even if not voted on) is also listed following a colon. For example, "(4/5:23)" indicates that the article has been viewed 23 times and voted on five times. Four of those five votes were positive, one of those votes was not.

Q. How do I rate an article?

A: At the top of every article, there is a box that gives the reader that ability to vote (thumbs up or thumbs down). Just click the appropriate button and your vote will be sent. After you have voted, the box will show the voting results for this article.

Q: I don't get the ability to rate an article. Why not?

A: We put in a half-hearted attempt to discourage duplicate voting. Note that this is not foolproof, nor is it intended to be. In truth it is more of a convenience to remind the reader if they have already voted on an article. Still, the method used relies upon cookies, and there are a few cases (such as cookies being disabled on the client) where voting will not work. We apologize for any inconvenience.

Writing Articles:

Q: Hey! I developed a neat programming trick and I'd like to share it with others. How can I get it posted in your guide?

A: Just send it in. Currently the policy of article addition to the database is wide open. Provided there is nothing offensive, profane, or plagiaristic in the article, it will be posted to the database in a reasonably short amount of time (i.e.: a few minutes to a couple of days).

Q: So am I limited to just "Hints and Tips" types of work or can I do other, bigger things.

A: Knock yourself out. You could write a series of large articles if you chose. Do whatever you think will benefit the community.

Q: Okay, what format should I put my articles in?

A: All articles are stored in the database in HTML format. This is the preferred form, but you can send the article in plain text if you would like. Although we will not modify the content of the article, in some cases it is possible that we may choose to modify the HTML tags for cosmetic reasons. If you are opposed to this, just let us know.

Q: Okay I wrote an article and have a couple of charts that accompany it. How do you handle that?

A: All article pictures will be stored in an "images" sub-directory named after the author (without spaces). For example, suppose that John Smith wrote an article which contains a picture named "NPCDiagram.jpg." The article's HTML should reference the URL:

```
/images/JohnSmith/NPCDiagram.gif
```

Q: Fine, in addition to pictures I also have sample code that is too long to put in the article. How can I include a sample file with my article?

A: Just like images, a directory will be created after the author's name under the "downloads" directory. Similar to the above example, links would look like this:

```
/downloads/JohnSmith/ExampleNPC.inf
```

Q: How do I send my stuff?

A: Just e-mail submissions to:

```
WebMaster@OnyxRing.com
```

Q: Okay, I wrote my article, and don't feel comfortable just "giving" it away. What protects me?

A: As the author, you retain all copyrights. Your name is always listed at the top of the article display page with a link to your e-mail address. At the bottom of the page is a courtesy copyright notice stating that you, as the author, retain all rights.

Q: Is there a problem with posting my article on other sites as well?

A: None, whatsoever. It's your article; do with it as you please.

Q. I've noticed the use of images for Drop Caps in some of the articles. Are these available for my articles as well?

A. Yes, although certainly not required, there is a set of drop cap images available for use in any article. The images are housed in the following directory:

```
images/dropdown1/
```

Each image is a three-letter JPG that begins with "dd" and ends with the letter of your drop cap. To use one of the site drop caps, use a variations of the following examples:

```
<img align="left" src="images/dropdown1/ddt.jpg">his is a story...
```

or

```
<img align="left" src="images/dropdown1/ddo.jpg">nce upon a time, there was a...
```

or

```
<img align="left" src="images/dropdown1/ddi.jpg">t was the best of times, it was...
```

Q. I would like to distinctively format code and game transcript samples. How can I do this?

A. Different authors choose to do this in different ways. The <PRE> tag has been set up with a mono-spaced font for exactly this purpose and some authors choose to use only that. Others may choose to take advantage of colors. Two CSS classes have been implemented for your use: "source" and "game." Also, because browsers implement colors differently, some authors find it best to utilize these

classes in a table, which wraps a <PRE>. The following HTML example will demonstrate:

```
<table class=source><td><pre>

print "hello world";
</pre></td></table><br>



<table class=game><td><pre>

&gt;x troll

She is ugly with green teeth. You cannot help but desire her.
</pre></td></table>
```

This displays as:

```
print "hello world";



>x troll
She is ugly with green teeth. You cannot help but desire her.
```

Part B
Backgrounders

What is IF?

F is an acronym for "Interactive Fiction." In the broadest sense of the term, "Interactive Fiction" is any work of fiction with which the reader/player interacts. To be reasonable, this could include visually oriented games like Diablo or Ultima. Even shoot 'em up games like Half-Life have a storyline to convey. But long before there were PCs to be taken for granted, back when "computers" meant "mainframes" that occupied entire rooms, and "cool graphics" meant correctly aligning the asterisks on the amber terminal screen, there were "Text Adventures".

For the purposes of this site, and the vast majority of the interactive fiction community, the terms "Interactive Fiction" and "Text Adventures" are synonymous. Text adventures, as the name implies, are programs done entirely in text. I use the qualification "programs" here to differentiate text adventures from written or static works of text. Examples of such works are books like the "Choose Your Own Adventure" series, and some more modern adaptations that exist in the form of web pages, today. Although some would argue that these are indeed "Text Adventures", they are outside the scope of what is generally being referred to when the phrase is being used. In standard text adventures, the player typically types out his intentions in plain English (or whatever other language the specific game is targeted for) and the games attempt to "understand" what was meant and perform the action.

Does that sound like AI? Not by today's standards, but in the Seventies, a "Natural Language" interface to a computer was unheard of. Yes, text adventures certainly seemed to bring reality several steps closer to Arthur C. Clark's 2001 vision of HAL.

The Baby IF...

What is generally accepted as the first-ever text adventure is a mainframe program written in the seventies, now known by the name "Colossal Cave". As is the case with most subculture-creating pieces of work, "Colossal Cave" birthed derivative works, such as "Adventure," which in turn served as a basis for many modern pieces of IF.

The Adolescent IF...

The company generally given credit for making huge advances in the interactive fiction evolution is a company named Infocom. After producing dozens of text adventures, Infocom went out of business (several years ago now) and sold the rights to their games to Activision. Strangely enough, years after the company's demise, Infocom works are still widely accepted as the defacto-standard in interactive fiction.

Today's IF (All Grown Up)...

Since Infocom closed it's doors, many people have put intensive work into furthering IF. Some have reverse-engineered the game files and executables of the old Infocom games and created a documented standard from them (Z-Machine). Because of this published standard, there are now dozens of interpreters that can be used to play Z-Machine games (including the old Infocom games). Interpreters exist for virtually any operating system, from a Linux workstation to a Windows CE PalmPC. A language and compiler has also been created (INFORM) and released as freeware. This compiler can be used to create original Z-Machine games and carry on the Infocom legacy.

In addition to the Z-Machine standard, other IF development platforms have also sprung into existence. The most accepted among these, which leads a close second to INFORM, is TADS. There are more as well, such as HUGO, ALAN, and ADRIFT. All of these can be located in the interactive fiction archive located at: ftp://ftp.ifarchive.org/if-archive/

Part C
Baby Steps

A Starting Point

here's a program that predates all other programs. It was written before Windows or X, before DOS or BSD, before even CP/M or TRS-DOS. It predates the PC entirely. You can find flavors of it written in every programming language in existence. From assembler to SQL, BASIC to C, Pascal to Fortran, Inform is no exception. Every programmer in the world has typed in a variation of this program. Some versions of it are complex. Others are simple. All share a common purpose: To provide the coder with a an introduction to a programming language. What is its name? "Hello World."

Several "Hello Worlds" exist for Inform already. Probably the most viewed of these was written by Graham Nelson, the author of the Inform language, and can be found in section 1.2 of his "Inform Designer's Manual." It is essentially what follows:

```
!This code was snatched from G. Nelson's
!Designer's Manual 3 sect 1.2
[Main;
     print "Hello World!^";
];
```

It is not all that inspiring, perhaps, but it does compile and when run, it does exactly what you would expect it to: It prints the words "Hello World!" on the display and then exits.

For a newbie Inform developer, there are several things that can be learned or deduced about Inform from the above example:

```
1) An exclamation point precedes comments.
2) A method or routine is enclosed in brackets and
     followed by a semicolon.
3) The method declaration (the name of the routine and
     parameters) is the first line of code inside the brackets.
4) Like C, all lines of code end in semicolons.
5) Also like C (there are numerous similarities between C and
     Inform), all Inform programs have acommon entry point
     method (or routine), named "Main."
6) Additionally, this program gives us a glimpse of a basic
     Inform routine called "print" which will output
     data to the display
```

Perhaps I expect too much from a "Hello World" program, but I also feel that there are also several things missing from that code that keep it from being a good starting place for 99% of developers (In its defense, the example was not really intended as a starting place. Mr. Nelson's finer example appears at the start of the

DM's third chapter).

```
1) The standard library is completely missing from the example.
2) There are not sample rooms for the player to be located in.
3) There are not sample objects for the player to interact with.
4) There is no prompt for the player to type in commands.
```

In theory, you could implement 2, 3, and 4 without implementing the standard library, but in reality you just don't. Why you ask? Simply put, the wheel is already in existence. There is no reason to invent a new one from scratch. The standard Inform library contains all things common among adventure games, including rooms, objects, and input routines. C coders can think of it as the C runtime library. A developer could choose to not use it and rewrite all the standard routines such as "printf," or "scanf" from scratch using the "_asm" keyword and low-level assembler code, but why? Even though they are different, C and the C library are essentially the same thing. The same is true for Inform and the Inform library.

So how do you reference the Inform library? Through includes. The inform library is just code. It exists in a collection of files and you include it by including the three main files in your source file via the include keyword (again, very much like C). These three files are "Parser.h", "VerbLib.h", and "Grammar.h" (on some machines the .h extension is dropped). Saddly, the placement and order of each of the files is significant and has an impact on whether or not your source code will compile.

The library looks for and expects certain things. Certain constants such as "Story" and "Headline" are used to define information about your game. Additionally, the Inform entry point has already been defined in the library. Now a routine named "Initialise" is expected for game initialization.

What follows is a "Hello World" program, which can serve as a starting point for coding any game. I personally keep a variation of this file stored in a "reuse" directory on my personal machine and copy every time I write a new game. Feel free to cut and paste it and change it to suit your preferences. Additionally, there is a slightly modified version of this that will serve as the starting point for many examples in later articles. It can be found here.

```
Serial "000001";
Release 1;
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Hello World
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! General defines
```

```
Constant Story "Hello World";
Constant Headline "^Copyright (c) 2001: Jim Fisher^";
#Include "Parser";
#Include "VerbLib";
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Maps
     object PlainPlace "Plain Place"
          with description "There really isn't
               much here. Just a room with no exits."
          has  light
     ;
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Movable Objects
     object     disc "small disk" PlainPlace
     with       name "small" "disk"
     ,          description "Engraved in this small disk
                are the words: ~Hello World!~"
;
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Code
[Initialise;
     location = PlainPlace;
];
#Include "Grammar";
end;
```

Part D
Intermediate Coding

Arrays, strings and other Inform goodies

he Inform Designer's Manual (4th edition) dedicates sections 2.4 and 2.5 to the world of arrays. From this point on, arrays crop up sporadically when it's necessary to explain how some function works or how to achieve some programming somersault. Like many other topics covered by the manual, you may find that while all the necessary information is there, the author's need for brevity (in a 576 page book that wastes not one single paragraph) leaves the novice with a feeling of "things a-happening beyond my grasp". This article aims to provide the novice to Inform with a little background material in order to more easily understand arrays.

Since the time when this article was written, Andrew C. Plotkin (aka. Zarf) has brought forth Glulx, a new Inform Virtual Machine which lifts the memory restrictions of the Z-Machine -- among other things. This made some parts of the text incomplete and some code examples "disasters waiting to happen". We have tried to bring the article up to date to cover Glulx compatibility and reduce disasters to the minimum. Even if you don't know what we are talking about, don't worry. Nothing that you read here will be hazardous to your games.

1. Basic concepts

An array is nothing more that an area in memory where a collection of data items can be stored sequentially, one after the other. Each item is called an "entry" in the array, and can be identified by the position it holds in the list. Example:

Position	**Item**
0	18
1	2354
2	127
3	4535
4	16000

The "position it holds" may be called "entry number" or "index" of the array, while the data "items" stored are simply "entries". Therefore we can say that, in the above example, the third entry is 127, at position/index/entry number 2. Note that in the example, the first index is 0. Some languages use 1 instead as the first index of an array. Inform uses both depending on the particularities of the array, as we

shall see.

Unfortunately, in order to understand how Inform manages arrays, it's useful to know a little bit about how computers store data in memory.

You are probably aware that the most common information unit in computers is the byte. One byte equals 8 bits, and since bits can take two values (0 or 1), there are 256 different bit combinations in each byte (2 to the power 8). This implies that if we want to store numbers in one byte, we can only handle quantities between 0 and 255. If we wish to store a single character, we can only code 256 different characters (which sounds good enough for an European alphabet). In fact, when storing characters, each one is encoded through a number (its ASCII code if it's from the American alphabet, or its ISO 8859-X code for other West and Central European alphabets -- which include local variations like the Spanish 'ñ'); that means that the item stored in the byte is always a number.

Computers frequently combine several bytes to store larger data items. For example, using two bytes together gives you 16 bits instead of 8 and, therefore, 65536 combinations (2 to the power 16). This is useful to encode characters from exotic alphabets which offer more than 256 symbols, like the Unicode standard, or for larger numbers (in the range from 0 to 65535). Computers and modern software may use more bytes (3, which means 24 bits; or 4, which implies 32 bits). This combination of bytes is called the "word size", so we may talk about a two-byte word size (16 bits) or a four-byte word size (32 bits).

Inform may compile for the Z-machine, which is a Virtual Machine (VM)designed by Infocom in 1.979 -- when memory was scarce --, or for Glulx, which is a new VM designed by Andrew Plotkin with Year-2000-Computers in mind. The Z-machine uses a two-byte word size while Glulx handles a four-byte word size. Most of your Inform code will compile fine for both (an appealing thought), but we must be careful when applying some calculations. We'll address this issue a bit later; simply bear in mind that when we mention two-byte word size or 16 bits in relation to the Z-machine, you may also read four-byte word size or 32 bits in relation to Glulx.

Inform -- which was originally conceived to recreate the Z-machine -- uses extensively this two-byte word size; in fact, every data type managed by Inform is stored in two bytes -- numbers, objects, some text strings, dictionary words... all of them are encoded in a 16-bit number and stored in a "word" (for those

conversant with 'pointers', let us say that everything in Inform are 16-bit pointers to memory addresses on the Z-machine). Single bytes are best used to store single characters.

When we wish to store an array in memory, we'll have to tell Inform if the items will be bytes (small numbers) or words (bigger numbers). For single characters, one byte should be enough, but if we have numbers or other data types in mind, words are much better. In fact, words may handle all kinds of data; the only advantage of a byte array is that it economises on memory.

2. Array declaration and usage in Inform

To declare an array, all we have to do is type the word "Array" followed by the name we wish the array to have and the symbol -> (for byte arrays) or --> (for word arrays). Finally, we indicate the number of entries (items) that our array will handle.

2.1 Byte Arrays

For instance:

```
Array four_bytes -> 4;
```

would declare an array with four entries, each of which will be a byte, since we have used the -> notation.

This generates a memory area with a size of 4 bytes, ready to store four small numbers.

Index	**Item**
0	0
1	0
2	0
3	0

IT'S VERY IMPORTANT TO REMEMBER THAT BYTE ARRAYS START TO NUMBER THEIR ENTRIES FROM 0

Let us suppose that we wish to assign the following items in the above array.

Index	Item
0	25
1	130
2	240
3	1000

The last number can't be stored in a single byte, since it goes beyond the 255 limit, so we're sure to run into trouble. We'll soon see what happens.

These numbers can be assigned to the array in the following manner:

```
four_bytes->0 = 25;
four_bytes->1 = 130;
four_bytes->2 = 240;
four_bytes->3 = 1000;
```

We've said that the last entry is too big for a single byte. This won't cause a complaint about the overflow problem at compile time, though. Inform will accept the array if its syntax is correctly written but, because of the way numbers are stored in bytes, our 1000 has been transformed into a different number. This will cause bugs at run-time that will be difficult to diagnose.

That's why number storage is better reserved for word arrays. We'll keep byte arrays just for single characters -- and they're only worth the effort if you really must save on memory. Let's try another example with the same array declaration.

```
Array four_bytes -> 4;
```

Index	Item
0	'F'
1	'i'
2	's'

3 'h'

Now we want to store the four characters which form the phrase "Fish". To accomplish this, we assign as follows:

```
four_bytes->0 = 'F';
four_bytes->1 = 'i';
four_bytes->2 = 's';
four_bytes->3 = 'h';
```

Observe the apostrophes -- or single quotation marks -- ('), as opposed to double quotation marks ("). Just like in C, a single letter surrounded by apostrophes is understood as "the ASCII code for this letter". (Note: If you put more than a single character between apostrophes, the compiler thinks you are defining a dictionary word. The use of single or double quotation marks is one of those syntactic niceties in Inform which may require another article, but until then, look up http://www.firthworks.com/roger/informfaq/ for the basics if you are interested).

We could now try to print these letters on the screen, perhaps with the aid of this loop:

```
for (i=0 : i<4 : i++)
  print four_bytes->i;
```

which means: set the variable i equal to 0; print the ith entry in the array and increase the value of i by one; repeat while i remains less than four.

We expect the phrase "Fish" to appear on the screen, but what we actually get is 70105115104. Hey, what's going on? Well, each of the 'F' 'i' 's' 'h' letters has been encoded in the array with a number (its ASCII code), 70 for 'F', 105 for 'i', 115 for 's' and 104 for 'h', and that's what the print statement has poured onto the screen, all glued together (print won't use spaces unless we tell it to.)

How do we effectively print the actual characters? We must politely indicate to the print statement that the item to output is a character (print is unaware of this, since we could have stored numbers in the array -- unless we state our wishes otherwise, print will always assume that we want to display a number.)

The correct loop would be:

```
for (i=0 : i<4 : i++)
  print (char) four_bytes->i;
```

And behold! We now succeed in getting Fish.

This way to assign data items to the array is a bit tiresome. Fortunately, Inform accepts a much more comfortable syntax to achieve the same result:

```
Array four_bytes -> 'F' 'i' 's' 'h';
```

Note that in this case we omit the number 4. Inform will count how many letters you wish to store in the array and will reserve the required space automatically.

There's even a shorter syntax for this:

```
Array four_bytes -> "Fish";
```

So instead of specifying each letter separately surrounded by single quotation marks, we put them all together between double quotation marks. Once again Inform counts four letters and reserves the required space for them.

In all of these examples, it has been the programmer's task to count how many letters there are in order to code the printing loop -- you have to indicate in the loop that the index goes from 0 to 3. This is tedious and error-prone (go on: make a byte array and a printing loop for "supercalifragilisticexpialidocious" if you don't believe it) and that's why string arrays exist.

2.2 String arrays (special byte arrays)

If we put "string" (with no quotes) instead of the -> symbol, we create a byte array with a special structure which happens to be quite useful.

```
Array a_few_bytes string 4;
```

This array declaration is just like the one we made before, but this time Inform creates a memory area for *5* bytes (even though we have asked for 4). The extra byte is used to store how many more bytes the array has. In our example, the extra byte would hold the number 4 to indicate that the array has four free entries.

Index	**Item**
0	4
1	0
2	0
3	0
4	0

This extra byte is always stored in the 0th entry of the index, and the remaining bytes will go between 1 and 4. ***IT'S VERY IMPORTANT TO REMEMBER THAT STRING ARRAYS NUMBER THEIR ENTRIES FROM 1*** -- unlike byte arrays, which started from 0.

Thus, if we want to store the letters "Fish" as in the previous examples, the tedious method of assignment would be:

```
a_few_bytes->1 = 'F';
a_few_bytes->2 = 'i';
a_few_bytes->3 = 's';
a_few_bytes->4 = 'h';
```

which of course may be shorthanded with the known variations, letter by letter...

```
Array a_few_bytes string 'F' 'i' 's' 'h';
```

...or "all together now":

```
Array a_few_bytes string "Fish";
```

With either "shorthand" method, Inform discovers that there are four letters and therefore stores number 4 in the 0th index entry of the array. This way the programmer doesn't need to know how many letters form the chosen phrase, because it can always be read from the 0th entry. The loop needed to print the stored letters would be:

```
for (i=1 : i<=(a_few_bytes->0) : i++)
  print (char) a_few_bytes->i;
```

Please notice that the variable i now begins with a 1 (and not 0, like before) and goes on while its value is less or equal than a_few_bytes->0 (the index entry that holds the length of the string). Number 4 has disappeared from the loop, which is now generic and can print strings of any length.

It's always a good idea to optimise code if you know how to. Suppose you have several arrays and you wish to print their characters: you could copy and paste the above loop and change the name of the array for each one of them, or you could create a general-purpose printing routine:

```
[char_array arr i;
  for (i=1 : i<=(arr->0) : i++)
     print (char) arr->i;
];
```

We now have a routine named char_array that defines two variables, arr (which will be fed with the name of the current array) and i. We could make a call to the routine from any point in our code, using either of these forms:

```
char_array(a_few_bytes);
```

or

```
print (char_array) a_few_bytes;
```

This way, the code necessary for printing characters stored in an array only happens once in the whole program.

2.3 Word arrays

Storing single characters has its uses, but the time comes when we wish to work with numbers, pointers (to functions, objects, messages) or dictionary words... which is all the same to Inform, since everything is encoded as 16-bit numbers (when compiling for the Z-machine) or 32-bit numbers (when compiling for Glulx). A byte array is not enough to handle these data types. What we need now is a word array.

Word arrays can be declared in the same manner as byte arrays, but we use the --> notation instead of ->.

```
Array four_words --> 4;
```

which creates a word array with four entries:

Index	Item
0	0
1	0
2	0
3	0

Now Inform, when compiling for the Z-machine, reserves an 8-byte memory chunk (because we want four elements, and each one of them requires two bytes).If, on the other hand, you compile for Glulx, the memory area needed will be 16 bytes (four bytes per entry).

To assign data items to the array entries, we do as before. First, the verbose method:

```
four_words-->0 = 25;
four_words-->1 = 130;
four_words-->2 = 240;
four_words-->3 = 1000;
```

You see that we now use --> instead of ->, because we are assigning words instead of bytes. We discover that we have no problem this time to store number 1000, because it fits nicely in a word.

We can define the array in one line, by using the following syntax:

```
Array four_words --> 25 130 240 1000;
```

Either way, we set up the array:

Index	**Item**
0	25
1	130
2	240
3	1000

We could display the contents of the array with a loop:

```
for (i=0 : i<4 : i++)
  print four_words-->i, " ";
```

And we'd get 25 130 240 1000.

As was previously the case with byte arrays, here the programmer must remember that the array holds four entries, in order to specify which is the maximum possible value of i in the above loop. This is a nuisance, so Inform provides a special word array (just as string arrays are related to byte arrays) which is called a "table":

2.4 Tables (special word arrays)

When we declare the word array, we can use "table" (no quotes) in place of the --> symbol. This generates a word array with a little extra structure which (again) turns out to be quite useful.

```
Array a_few_words table 4;
```

In this case (analogous to the string array), Inform reserves an extra entry in the array (so that this array now needs 10 bytes = (4+1)*2 for the Z-machine or 20 bytes = (4+1)*4 for Glulx) which holds the number of entries minus itself -- so in the example, a_few_words-->0 would hold number 4).

Index	Item
0	4
1	0
2	0
3	0
4	0

IT'S IMPORTANT TO REMEMBER THAT TABLES BEGIN TO NUMBER ENTRIES FROM 1 because the 0th entry is reserved to hold the length of the array.

Example:

```
Array a_few_words table 25 130 240 1000;
```

This too would create an array with five entries.

Index	Item
0	4
1	25
2	130

3 240
4 1000

Now let's display the contents of the table. We need another loop:

```
for (i=0 : i<=(a_few_words-->0) : i++)
  print a_few_words-->i, " ";
```

3. Summary

The following table summarizes the different kinds of arrays, how to access their entries and the allowed range for their entries. If the programmer ignores this range and breaks the bounds of the array (by calling a->5 in a byte array with four entries, for instance) the compiler will remain silent, but at run-time we'll get a glorious crash.

Declaration	Size of each entry	Access to entries	Entries range
Array abc -> 4	byte	abc->i	0 ... 3
Array abc string 4	byte	abc->i	0 ... abc->0
Array abc --> 4	word	abc-->i	0 ... 3
Array abc table 4	word	abc-->i	0 ... abc-->0

4. Basic concepts with a twist

4.1 String arrays are NOT strings

This is yet another one of those things that leads programmers into confusion.

A string array is a sequence of single characters stored in memory one by one (each character takes one byte). You can access any single character through its entry number. For instance:

```
Array abc string "I like fish";
```

If we take a look at the value stored in abc->4, we'll get the fourth letter from the text, in this case 'i' (well, we'd get its ASCII code, which is 105). In order to know the length of the text, all we have to do is check abc->0.

No surprises so far. Everything behaves as it has been explained. C programmers nod happily in understanding.

However, storing text through the use of arrays has a couple of snags:

```
1) They consume quite a lot of memory. A 1000 character text would take 1000 bytes.
Since text adventures are basically made of text, it would be desirable to store
text more sparingly.


2) Texts stored in arrays are visible in the compiled game file. If anyone cares to
take a look at the .z5 file (with any editor, even the ancient TYPE from DOS), "I
like fish" will stand out among the incomprehensible encrypted characters. An
impatient player could find hints and messages from the story file. If you do not
wish that to happen, we need a way to encrypt all texts so that unwanted peeking at
the story file reveals nothing.
```

That's why Inform encodes most of the texts in a way that compresses their size, with a rough 2/3 ratio, and makes them gibberish without the appropriate Z-machine interpreter.

This way of storing text is what Inform calls "strings", which is, perhaps unfortunately, the same word used for "string" arrays, even if the meaning is quite different. In this article, we shall call them "encrypted" strings as opposed to "conventional" strings (which will stand for string arrays).

Any text in double quotes, "this one, for instance", is encoded by Inform as an encrypted string, unless it's part of a string array declaration, in which case it will be encoded as a conventional string. Example:

```
Array abc string "I like fish";

Global pqr = "But I don't like fishing.";

Object river "gurgling river"
  with description "It's full of fish.";
```

In the first case, the string "I like fish" is encoded as a conventional string, letter by letter, with no compression or encryption. Each character may be read through its entries: abc->1, abc->2, etc.

In the other two examples, text is encoded as encrypted strings. They take up less memory and are unreadable by curious eyes. On the other hand, it is impossible to

read each character separately or to know the length of the encrypted strings, because they are not arrays. Once the compiler has encrypted them, their values get stored in the variable pqr as 16-bit numbers (with no apparent meaning).

If we try to

```
print pqr;
```

we'd get that meaningless number. If we want to "decipher" the encrypted string and discover the text, we have to use a slightly different syntax:

```
print (string) pqr;
```

Now we get "But I don't like fishing." on the screen. This syntax, however, will not work for abc, because abc is not an encrypted string. If, nevertheless, we try:

```
print (string) abc;
```

the result will be unpredictable, because print (innocently unaware of our mischievous intent) will try to decipher the contents of abc as it were an encrypted string. The way to get at abc has already been shown; it requires a loop:

```
for (i=1 : i<=(abc->0) : i++)
  print abc->i;
```

Object names and descriptions are encrypted strings too, but it's the library that takes care of printing them when necessary -- using print(string), naturally.

4.2 Dictionary words are NOT strings

Inform builds up a dictionary with the words that the game will "understand" when the player types them. For instance, the words given in the <name> property of an

object or the defined verbs. The dictionary will gather all the words surrounded by single quotation marks that we write in our game. For instance:

```
Object piranha "piranha"
  with name 'piranha',
  ...
```

Why do we repeat piranha so much? The first piranha -- with no quotes -- is the inner name of the object. It will only be seen and used by the programmer when he/she wants to refer to the piranha object. The second "piranha" -- in double quotes -- is the word that the player will see when the game needs to mention the piranha object (say, in an inventory). It's an encrypted string. The third 'piranha' -- in single quotes -- is the word that the player will be able to type when he/she wishes to interact with the piranha. This will go in the game dictionary.

CAUTION: Inform allows you to put double quotes around the third piranha and it will still be understood as a dictionary word. This only happens in the <name> property of an object and in the <verb> directive -- otherwise, anything you write for properties between double quotes will be understood as an encrypted string. DO NOT USE THIS UGLY AND CONFUSING PRACTICE. The clearer the syntax, the clearer the concepts.

A dictionary word may be stored in any global variable, like this:

```
Global xyz = 'shark';
```

In this case, 'shark' is neither a conventional string nor an encrypted string. Inform transforms it into a number (which is the place the word holds in the dictionary). If we try to:

```
print xyz;
```

we'd just get a meaningless number. If we try:

```
print (string) xyz;
```

the result will be chaos, because print will try to decipher the dictionary number like it were an encrypted string (which it isn't). We can't read the individual characters in 'shark' because it's not a byte array. The only way you could eventually output the word 'shark' would be through:

```
print (address) xyz;
```

The (address) modifier tells print to display the dictionary word whose number we are providing.

4.3 Arrays as object properties

Most objects have a property which is, in fact, an array -- the "name" property. This is an array whose entries will be the dictionary words that the player may use when interacting with the object. Consult the end of section 3.5 in the Designer's Manual. Another example of an array used as a property is found_in, which lists all the locations you want a certain object to be in (useful for coding general scenery fixtures like bushes, carpeting, or even the sun -- if it's visible from various places).

When arrays are used as properties and not as global variables (which was the case of everything explained in 2), the syntax is different, because you don't need neither the directive Array, nor any of the -> or --> symbols, nor the modifiers "string" or "table". For instance:

```
Object strange_thing
  with a_few_words 25 130 240 1000;
```

We declare an object called strange_thing. This is not your usual text adventure object, because it lacks name and description, but from a programming point of view, it's a perfectly valid object (Inform will compile it without a whimper).

This object has just one property called a_few_words (it's perfectly legal to create our own properties, as explained in section 3.5 of the DM). It could store a number

or a collection of them. The difference resides in what we write after the property name.

In the above example we write four numbers (separated just by spaces... no commas, semicolons or any other punctuation), so Inform will understand that a_few_words is in fact a collection of four items, with values 25 130 240 1000. These collections will always become word arrays (it is actually impossible to make a byte array out of a property), but this is hardly a limitation, since almost always we'll want the property to hold 16/32-bit items. The question now is: how do we read the entries of these arrays?

Let's try an apparently reasonable statement:

```
print strange_thing.a_few_words;
```

This will print just the first entry of the property (in the example, number 25). This would be all right if the property held just one value, but since we have more, we might need to know: how many are there? and: how do we access each one of them?

The answer is simple, but the syntax is a bit ugly. The value:

```
strange_thing.#a_few_words
```

tells us how many bytes are there in the property (its "length"). We now reach the place where all the previous ramblings about Z-machine's two-byte word size and Glulx's four-byte word size become important.

Since the information we get from the value is the total number of bytes taken by the whole property, in order to discover how many entries we've got, we need to divide this global quantity by the number of bytes that store a single entry.

Remember that in the Z-machine, each entry from a word array was stored in two bytes, while in Glulx it took up to four bytes. So it seems that, depending on the VM you are planning to use, you should divide either by 2 or by 4 (this is true, and if you divide wrongly you'll get in trouble). Fortunately, there is a way around this, as

we shall soon see.

Now, the value:

```
strange_thing.&a_few_words
```

is a word array which collects the numbers 25 130 240 1000 (remember that the entry numbers begin with 0).

Suppose we want to print these numbers on the screen. On the days when we only had Z-machine, we needed to code a loop thus:

```
for (i=0 : i<(strange_thing.#a_few_words/2) : i++)
  print strange_thing.&a_few_words-->i, " ";
```

For Glulx, you should change

```
i<(strange_thing.#a_few_words/2)
```

for

```
i<(strange_thing.#a_few_words/4)
```

But what if you don't know what platform you'll be using? Well, you can use cautious code for the undecided. There is a bi-platform library and a bi-platform compiler for Inform so that you may compile your source code for either VM -- and that's what you'll be using if you want to compile for Glulx. This compiler predefines a needful constant named WORDSIZE: if you choose to compile for the Z-machine, its value is set to 2, but if you compile for Glulx it will be set to 4. You may then code:

```
i<(strange_thing.#a_few_words/WORDSIZE)
```

and now the problem is almost out of the way.

Then again, you may not be sure yet if you'll want your game compiled with Inform's Z-machine compiler (the original stuff written by Graham Nelson, which is widely used and much beloved). The current version of Graham's compiler (6.21) does not define WORDSIZE, so you need to add the following lines at the top of your code, before any "Includes":

```
#ifndef WORDSIZE;
Constant TARGET_ZCODE;
Constant WORDSIZE 2;
#endif;
```

TARGET_ZCODE is another constant predefined in the bi-platform compiler (if you are beginning to feel fascinated by these concepts, feel free to consult Zarf's "The Game Author's Guide to Glulx Inform", http://www.eblong.com/zarf/glulx/inform-guide.txt). The only snag is that we get a compiler warning, since we define a constant -- TARGET_ZCODE -- that is not used. You can avoid it (logically) by forcing a harmless use of the constant. If you write:

```
#ifndef WORDSIZE;
Constant TARGET_ZCODE 0;
Constant WORDSIZE 2 + TARGET_ZCODE;
#endif;
```

you make TARGET_ZCODE equal to zero and add it to WORDSIZE. The constant is in use (although altering nothing) so the compiler is happy. And, much more importantly, our code will be ready to accept either set of Inform libraries or compilers.

Back to properties. If the property were to hold encrypted strings or dictionary words, we would still need the appropriate modifiers (string) or (address). For instance:

```
Object stone "stone"
  with name 'pebble' 'stone' 'rock',
     description "It's an ordinary stone.";
```

If we wish to list on the screen the stock of dictionary words that the player can use to refer to the stone, we would code:

```
for (i=0 : i<(stone.#name/WORDSIZE) : i++)
  print (address) stone.&name-->i, " ";
```

If we only want to show the first of these words, we just need:

```
print (address) stone.name;
```

And to show the description:

```
print (string) stone.description;
```

This is because there's only one element in the description property. There could be more than one (theoretically), in which case we would proceed thus:

```
for (i=0 : i<(stone.#description/WORDSIZE) : i++)
  print (string) stone.&description-->i, " ";
```

I can hear you thinking "Why on earth am I going to need to list the dictionary words of an object or its highly improbable multiple descriptions?", and you are right, you'll probably never need to do so. However, you may define sometime a property which holds various values and, if you want to access them, this is how it's done. Understanding how things work and the philosophy of the language will bear fruit some day.

That's it for now. Remember that a good practice is to borrow shamelessly other

people's code and to try to modify it, to see if things work as we expected.

5. What do I do now?

"That's a lot of handsome theory, but I didn't come here for a lecture. I want to program my own games and I want to know the practical uses of arrays right now!"

Well, there's a couple of places you might want to check. Firstly, go back to the DM, make a search for "array" and start looking at the examples. There's quite a lot of them and you should be able to tell by now how the array bits fit in the puzzle. Also, Marnie Parker's Inform Primer (http://members.aol.com/doepage/doefaq.htm) covers in section 8 a few interesting working examples of array usage.

Code Reuse: Building a Personal Code Library

Reusable code and reasons to implement a "personal code library"

eusable code is the bomb. Every programmer that has been programming for any length of time has reusable code somewhere. In a perfect world that code would be easily accessible and generic enough to simply drop into an existing work in progress. Sadly, all too often, developers find themselves sorting through previously written source files to refer to some algorithm or implementation coded years ago but again needed. Once found, "cut and paste" is usually followed by lengthy modifications that are required to get the old code to work in the new program.

At the most basic level, this "copy and modify" method still qualifies as reusing code. Still, while virtually every developer at one time or another has used this technique, it leaves something to be desired.

A far better way of reusing code is to implement a personal code library. Done correctly, it allows for several advantages over non-library techniques:

```
1) Code naturally tends to be more organized and programs
     become modular in nature.
2) Reusing existing code speeds up development.
3) Fewer bugs can be introduced if less coding is done so
     new projects are naturally more stable.
4) When bug fixes are found and fixed in the library, they
     are automatically fixed in all other works that share
     the library.
5) Shared code gives a common feel across all of your works
     giving them a sort of "signature."
```

It should be noted that there are several different ways of reusing code. The techniques shown in this article have worked best for me, but the reader may prefer different techniques. Keep an open mind, because my way is not the only way.

Three steps to a personal code library

By just a few simple steps, a developer can build a library of customized code that can be drawn upon with a few simple keystrokes. The first step, we've already touched upon....

Step one, template code, is really the basis for any serious library. The practice of making code templates consists simply of identifying code that you use more than once (or are likely to use again) and saving it in a special file so that

it can be found more easily. A blank game is a prime example of a code template (see "A Starting Point.")

Additionally, patches to the standard Inform Library, generic methods (like the Center routine), and common classes from which specialized objects are inherited are all good candidates for your library. Anytime a developer begins to search through old code to copy into a new project, a flag should go up. This is an indication that the code being tracked down is a good candidate for a template. Once isolated, and the code is placed in its own file, it can be more easily found and copied into a new project. Taking the time to identify what code is really reusable is half the battle and will pay for itself in both organization and rapid development.

There are con's associated with using this approach alone, however. While cutting and pasting organized template code into a new project may speed up development, it lends itself to an unorganized project. This method alone has a tendency to begat spaghetti-code and special care needs to be taken to organize the destination source file. Additionally, when a bug has been identified in a template, it must be fixed in all projects in which the template was pasted. Luckily, Inform hosts a directive that minimizes our need to "cut and paste."

Step two, the practice of including modules has several distinct advantages that augment the power of template coding and address the irritations associated with "cut and paste." The most notable advantage is shared code. Implemented via Inform's #include directive, the same template can be referenced in multiple works. Modifications to the shared file affect all referencing projects. A bug fixed in one, is fixed in the others. This includes programs in which the bug has yet to be identified. This leads to more stable and robust code throughout all of your works.

Again, there are some annoyances with this technique. Because of the nature of the Inform compiler and the standard library, most template code cannot simply be added to a game file in one large block (which is essentially all the #include directive does.) Hidden in the back of "The Inform Designer's Manual" Appendix 11, are directions concerning what goes where in an Inform Program. From this, you can derive four sections (surrounding the standard library's #include's) where library code needs to be placed:

```
1) The REPLACE section - appears prior to the inclusion of
     Parser. REPLACE directives, used when modifying system
     routines, go here.
2) The MESSAGE section - appears between the Parser and Verblib
```

```
     includes. Definitions of the LibraryMessages object,
     SACK_OBJECT, and the task_scores array goes here.
3) The CODE section - appears between the Verblib and Grammar
     includes.  Attribute, Property, Classes, and Object
     implementation goes here. Most implementation code is
     placed in this section.
4) The GRAMMER section - appears following the Grammar include.
     All new Verb and Extend directives go here.
```

One possible, though not recommend, way of addressing this is by implementing separate files for each section needed by a library entry and including them at the appropriate location. Obviously this file juggling can be managed in a better way. Again, Inform offers a better solution.

Step three, using conditional compilation, is the final technique needed to bring everything talked about thus far together. Inform implements conditional compilation with various #If... directives. While the #include directive adds code to a file where none actually exists, these directives give the ability to ignore code in a file that does exists. Using these directives, the compiler can selectively dismiss sections of code and a large, diversified library can still produce a tight specific program.

Creative use of these directives can cause a file to be interpreted differently each time it is compiled. This allows us to address the hassles identified above but coding a module to be interpreted differently depending on where in the program it is included. As an added bonus, we can wrap all library includes into a "single point of entry" include file and choose which library packages we include by declaring constants. This makes for even cleaner code. Not bad.

The Framework and the Implementation

To the experienced developer, all of this coding with conditional directives means one thing: a framework. Obviously the content of your personal library will vary from developer to developer, but the implementation framework has already been coded and is available to all. It contains everything you need to begin developing and using a personal library. That is, a sample blank game template (implementing the library), a library entry template, and a "single point of entry" file. Additional library entries are also available for individual downloads, or a complete package of every thing is also available. All of this for the great low price of a download click from http://www.onyxring.com/orlibrary.aspx.

As always, the best way to learn is to look at the code. There are several comments inside each of the files that describe how to use it.

Happy coding.

Print rules

This topic is about using Inform's **print** statement. Everything said here for **print** applies also to **print_ret**-- remember that these are exactly equivalent:

```
print stuff_to_be_printed; new_line; rtrue;
print stuff_to_be_printed; print "^"; return true;
print_ret stuff_to_be_printed;
```

The **print** statement can be used like this:

```
print term;
print term;
...
print term;
```

However, since it's just about the only statement where commas play a significant role, you'll more often see it in this form:

```
print term, term, ... term;
```

where you must have at least one *term* and can have as many as you like. Inform copes automatically with literal numbers and strings; this example outputs the expected "2 plus 3 is equal to 5." result:

```
print 2, " plus ", 3, " is equal to ", 2+3, ".^";
```

Things get more interesting as soon as you start to use variables. Consider these two examples:

```
x = 2+3;
print 2, " plus ", 3, " is equal to ", x, ".^";

x = " is equal to ";
print 2, " plus ", 3, x, 2+3, ".^";
```

The first still works fine, but the second produces the somewhat baffling "2 plus 3197685." which isn't at all what you want to find; between the "3" and the "5", where you'd hoped to see the phrase " is equal to ", you've instead got the mysterious number 19768. What's going on? Simply this: each Inform variable is stored in a single computer word -- that's just sixteen bits (or thirty-two bits in

Glulx) -- fine for holding a number such as the result of adding 2 and 3, but far too small to hold a long string of characters like " is equal to ". Instead, the Inform compiler places those characters in a special memory area dedicated to holding all of the game's strings, and stores a pointer into that area -- the string's **address** -- in the variable. That's what "19768" is: the memory address where the phrase can be found.
So, how do you print what you want: the string itself rather than its fairly meaningless memory address? Like this:

```
x = " is equal to ";
print 2, " plus ", 3, (string) x, 2+3, ".^";
```

The general message coming out of all this is that, unless told otherwise, Inform treats each variable value simply as a number to be printed in signed decimal format. If you know better -- that the value has to be printed in some other format -- you need to explicitly say so. Typical cases are where:

- the value isn't actually a number as such, but rather an address of an Inform item such as a string, object, or word in Inform's dictionary;
- the value *is* a number, but you don't want it printed in signed decimal format, but instead in words, or as a hexadecimal value, or whatever;
- the value is an encoded representation of some data which needs to be decoded or transformed before being printed.

Built-in rules
The way that you say to Inform "treat the following value specially, rather than just print it as a decimal number" is by using a **rule**: a word in parentheses immediately before the value. That's what **(string)** is -- a rule that tells Inform to treat what follows as the address of a string to be printed. Several built-in rules are available:

(number)*expr*	the expression's value in words [1]
(char)*expr*	the expression's value as a single character
(string)*addr*	the character string at this address [1]
(address)*addr*	the dictionary word at this address
(name)*addr*	the name of the object at this address
(a)*addr*	the object's name preceded by "a/an/some" [2]
(the)*addr*	the object's name preceded by "the" [3]
(The)*addr*	the object's name preceded by "The" [3]
(ItorThem)*addr*	"him/her/it/them" [4]
(ThatorThose)*addr*	"him/her/that/those" [4]
(CThatorThose)*addr*	"He/She/That/Those" [4]
(CTheyorThats)*addr*	"He's/She's/That's/They're" [4]
(IsorAre)*addr*	"is/are" [4]

Note 1: There's an important distinction between an *expr* and an *addr*: you can perform arithmetic in the former, not in the latter. **(number) 2+3** and **(char) 'q'-1** will both work, but **(string) "Christmas"-5** and **(name) self_obj+8** are recipes for disaster.
Note 2: Unless the object has an **article** property or a **proper** attribute.
Note 3: Unless the object has a **proper** attribute.
Note 4: The appropriate value is determined the attributes of the object at this address.

Homegrown rules

The built-in print rules are very useful, but there's nothing to stop you adding extra ones of your own. All you need do is write a routine of the desired name, taking a single parameter which is the value to be printed. The Inform Designer's Manual (Edition 4/1 §1.12) provides a sample `hex()` routine so that you can, for example, **print (hex) -4** to produce the output "fffc". (Caution: don't use the equivalent routine from the DM Edition 3 -- it mishandles negative numbers.) Also, `Card()` in §2.4 decodes a number 0..51 into a playing card. Here are a few other examples.

Possesive pronouns. Like **(a)** and **(the)**, to print "his/her/its/their" to suit the specified object; see also DM 4 Exercise 62:

```
[ hisorher x;
        if (x has pluralname) print "their";
        else if (x hasnt animate || x has neuter) print "its";
            else if (x has female) print "her"; else print "his";
        ];
. . .
print "In the bed dozes ", (a) NPC1, ", twitching and muttering under ", (hisorher)
NPC1, " breath.";
```

Non-zero numbers. A version of the standard **(number)** rule which print "no" instead of "zero":

```
[ nznumber x; if (x > 0) print (number) x; else print "no"; ];
. . .
print "You have ", (nznumber) coins.number, " coins left to spend.";
```

Typographic control. Switching into boldface or underlined/italicized type normally requires a separate **style** statement. Here, you can control the typography (of a string) within a **print** statement:

```
[ b x; style bold; print (string) x; style roman; ];
[ u x; style underline; print (string) x; style roman; ];
```

```
. . .
print "...and can print in ", (b) "bold face", " as well as using ", (u) "italics",
"."; 
```

Decoding. Sometimes, you work internally with encoded values which need to be turned into string representations only at the instant of being printed. For example, a day-of-the-year (1..365) is transformed into a meaningful date (no, it doesn't handle leap years):

```
[ dayofyear x; switch (x) {
          1 to  31: print (ordinal) x,     " January";
         32 to  59: print (ordinal) x- 31, " February";
         60 to  90: print (ordinal) x- 59, " March";
         91 to 120: print (ordinal) x- 90, " April";
        121 to 151: print (ordinal) x-120, " May";
        152 to 181: print (ordinal) x-151, " June";
        182 to 212: print (ordinal) x-181, " July";
        213 to 243: print (ordinal) x-212, " August";
        244 to 273: print (ordinal) x-243, " September";
        274 to 304: print (ordinal) x-273, " October";
        305 to 334: print (ordinal) x-304, " November";
        335 to 365: print (ordinal) x-334, " December";
        default:    print "BAD DATE";
        } ];
[ ordinal x; print x; switch (x) {
        1, 21, 31:  print "st";
        2, 22:      print "nd";
        3, 23:      print "rd";
        default:    print "th";
        } ];
. . .
print "Today's date is ", (dayofyear) somedate, ".";
```

Monetary values. Suppose your game involves cash transactions -- acquiring and spending 'real' money rather than notional zorkmids or gold coins. You might choose to store a monetary value in a variable, for example in units of cents or pennies; that way, you can easily add to and subtract from your cash float. Here's a rule which prints such as variable as a monetary value (for example, it prints 1234 as $12.34). Because you can't have negative cash amounts, it treats the variable as an unsigned value in the range 0..65535 ($0.00 to $655.35). If you need to deal with larger amounts, you're on your own.

```
[ cash x
        a b c d e;
        if (x < 0) {
            x = x & $7FFF; e = x%10 + 8; x = x/10 + e/10 + 3276; e = e%10;
            }
        else { e = x%10; x = x/10; }
        d = x%10; x = x/10; c = x%10; x = x/10; b = x%10; a = x/10;
        print "$";
        if (a) print a;
        if (a || b) print b;
        print c, ".", d, e;
        ];
. . .
print "Tallying your possessions, you find you have ", (cash) player.cashfloat, "
in ready money.";
```

Discarding return values. Consider this (erroneous) example, where you invoke a routine which produces its own output:

```
[ dow d m y
        a b;
        ! Algorithm from http://www.tondering.dk/claus/calendar.html
        a = (14-m) / 12;
        b = (y-a);
        a = (d + (m+(a*12)-2)*31/12 + b + b/4 - b/100 + b/400) % 7;
        switch (a) {
            0: print "Sunday";
            1: print "Monday";
            2: print "Tuesday";
            3: print "Wednesday";
            4: print "Thursday";
            5: print "Friday";
            6: print "Saturday";
            }
        ];
. . .
print "Christmas Day falls on a ", dow(25,12,2001), " this year.";
```

What happens, when you run this, is that the dow() routine generates its output and returns, whereupon the **print** statement outputs the value returned by the routine, usually "1" (representing **true**). There are several ways of preventing this spurious "1" from being printed. You can split the **print** into its component statements:

```
[ dow d m y; ... 6: print "Saturday"; } ];
. . .
print "Christmas Day falls on a ";
dow(25,12,2001);
print " this year.";
```

Alternatively, you can (sometimes) rewrite the routine to return a value rather than printing it:

```
[ dow d m y; ... 6: return "Saturday"; } ];
. . .
print "Christmas Day falls on a ", (string) dow(25,12,2001), " this year.";
```

However, easier than either of these is simply to create a rule which absorbs and ignores the unwanted value returned by the routine:

```
[ dow d m y; ... 6: print "Saturday"; } ];
[ ignore; ];
. . .
print "Christmas Day falls on a ", (ignore) dow(25,12,2001), " this year.";
```

Be sure that you're comfortable with the difference between **print (someroutine) x;** and **print someroutine(x);** -- they both call someroutine() with x as a parameter, but whereas the output produced by the former is only what (if anything) the routine itself *prints*, the latter additionally outputs the value *returned by* the routine, which

is often "1".

BITWISE AND LOGICAL OPERATORS IN INFORM

n a programming language like Inform, the *operators* are those little symbols such as + - * / = which combine numbers and variables to form *expressions*. For example, when you see something like:

```
average = (x + y)/2;
```

you're looking at an Inform statement which uses an assignment operator "=", an addition operator "+" and a division operator "/", plus a pair of parentheses (...) to ensure that the addition happens before the division. Even if the word "operator" itself isn't familiar to you, the role of arithmetic symbols like these shouldn't come as much of a surprise. Likewise, you're probably fairly comfortable with conditional operators such as ">=", which tests if one value is greater than or equal to another value:

```
if (average >= 7) print "Well done!";
```

Sections 1.6 through 1.8 in the Inform Designer's Manual introduce the operators which are most frequently found, and Table 1 provides a full list. As well as the commonplace arithmetic and conditional operators, there are others whose behaviour is sometimes less intuitive. This article focuses on two families which are easily confused -- *bitwise operators* and *logical operators* -- and explains their different uses.

1. Binary Numbers

Before getting started on the operators, we need to touch briefly on how data is stored. Computers munch down all information thrown at them and finally convert it into Binary, a base-2 numbering system of 0's and 1's. Binary represents electrical impulses either in an ON or OFF state, just like switches. The computer deals only in the flow of impulses; humans need to devise conventions so that certain combinations of these two digits represent numbers, letters, pixels, musical tones or any other kind of data.

When you type a number as part of a program, Inform assumes that you've typed a *decimal* (base-10) number unless you include a special prefix. There are two of these: "$" introduces a *hexadecimal* (base-16) number, and "$$" introduces a *binary* (base-2) number -- see section 1.4 of the DM. For example, here are the three ways of typing the number of years in a century:

```
100    $64    $$1100100
```

All of those have the same value (though the first form is the most common and natural-looking), and they're all stored in the computer in the same way (in binary -- the third form). As another example, the three numbers:

```
100    $100    $$100
```

are in everyday decimal worth respectively: one hundred, two hundred and fifty-six, and four. In this article, we won't mention hexadecimal numbers again, and we'll use "$$" to denote a binary number.

Inform works both with 16-bit numbers (for the Z-Machine) and 32-bit numbers (for Glulx) -- "bit" stands for BInary digiT. This means that each of your numbers is converted to a sequence of either sixteen or thirty-two bits, each bit having a value of 1 or 0.

A 16-bit number (Z-Machine):

```
$$1001100011100101
```

A 32-bit number (Glulx):

```
$$00001100111010010011100110101100
```

2. Bitwise Operators

Working directly with binary values is cumbersome; fortunately, you rarely need to. There are occasional situations, however, when programmers want to (or have to) work with low-level data, and this is where the bitwise operators make an appearance.

Inform defines three bitwise operators, which enable you to manipulate the individual bits in a number:

&	bitwise AND operator	merges two numbers
bitwise OR operator	merges two numbers	
~	bitwise NOT operator	changes one number

The & (AND) operator compares and merges two numbers bit by bit. For each of the 16 (or 32) comparisons, the output bit at that position is set to 0 unless both bits being compared are 1, in which case it too is set to 1 (that is, an output bit is set to 1 if the first input bit AND the second input bit are 1). So:

$$1010 & $$1100 evaluates to $$1000

(Note: You may find this clearer if you write it with the equivalent bits vertical aligned:

$$ 1 0 1 0 &
$$ 1 1 0 0
---------- evaluates to
$$ 1 0 0 0

where you can see that both input bits are 1 only in the leftmost case.)

The | (OR) operator compares and merges two numbers bit by bit. For each of the 16 (or 32) comparisons, the output bit at that position is set to 1 unless both bits being compared are 0, in which case it too is set to 0 (that is, an output bit is set to 1 if the first input bit OR the second input bit is 1). So:

$$1010 | $$1100 evaluates to $$1110

The ~ (NOT) operator inverts all of the bits of a number, changing each 0 to 1, and 1 to 0. So:

~$$1100 evaluates to $$0011

(Note: We are using small numbers for the sake of clarity. $$1100 should be read as $$0000000000001100 on the Z-machine,

$$00000000000000000000000000001100 on Glulx. This would only affect -- in the examples -- the NOT result, where all 0's to the left of our defined number turn into 1's.)

So far, so good; you can study the binary bit patterns and see what's going on. But remember, it's perfectly valid to use a bitwise operator with decimal numbers:

100 & 20

You have to bear in mind that the operator always works on the bit patterns of the decimal numbers. If you were to read this expression using normal English, you might say "100 AND 20", which perhaps sounds like the result would be 120. But:

$$1100100 & $$0010100 evaluates to $$0000100 (4 in decimal notation)

Also, 100 | 20 is worked as $$1100100 | $$0010100, which evaluates to $$1110100, 116 in decimal notation.

You can demonstrate this using the example program in section 1.4 of the DM, with a small variation:

```
[ Main x y;
  x=100; y=20;
  print "Today's number is ", x & y, ".^";
];
```

This outputs:

```
Today's number is 4.
```

The basic print statement always outputs decimal numbers. To see the result in binary, you might use:

```
[ Main x y;
  x=100; y=20;
  print "Today's number is ", (bin) x & y, ".^";
```

```
];
[ bin x i;
  print "$$";
  for (i=16 : i>0 : i--) {
    if (x < 0) print "1"; else print "0";
    x = x + x;
  }
];
```

In theory, you can apply bitwise operations to lots of thingies, if you understand their binary representation. In practice, when coding a normal Inform game you'll probably be using bitwise operators with the same frequency as Mandarin Chinese...

3. Logical operators

Whereas the bitwise operators are fairly rare, you'll frequently encounter the logical operators, usually in conditional statements like **if** and **while**. Happily, they're also easier to understand, since their only concern is whether a number is **false** (that is, zero) or **true** (any non-zero value). For this section, we don't need to worry about binary bit patterns.

There are three logical operators:

&&	logical AND operator	tests two numbers
||	logical OR operator	tests two numbers
~~	logical NOT operator	tests one number

In each case, the result of the test can only be **false** (0) or **true**(1).

The && (AND) operator determines whether both numbers are **true**. So:

```
x && y
```

evaluates to 1 if x is **true** and y is **true** , and otherwise to 0. Note that the tests are performed left-to-right: if x turns out to be **false**then the result is bound to be 0, so the game doesn't bother testing y.

The || (OR) operator determines whether either number is **true** . So:

```
x || y
```

evaluates to 1 if x is **true** or y is **true** , and otherwise to 0. Note that the tests are performed left-to-right: if x turns out to be **true**then the result is bound to be 1, so again the game doesn't bother testing y.

The ~~ (NOT) operator inverts a number, changing **true** to false and false to **true**. So:

```
~~x
```

evaluates to 0 if x is **true** , and otherwise to 1.

(Note: You should be aware of the way Inform handles the precedence of either bitwise and logical NOT operators. Please check http://www.firthworks.com/roger/informfaq/ for more information).

Here again is the example program in section 1.4 of the DM, now showing a logical operation:

```
[ Main x y;
  x=100; y=20;
  print "Today's number is ", x && y, ".^";
];
```

This outputs:

```
Today's number is 1.
```

and if you change "y=20;" to become "y=0;", you'll get:

```
Today's number is 0.
```

The examples in this section have shown the logical operators testing the values of variables x and y. More commonly, though, you find them working on the outcome of the conditional operators. For example:

```
(x > y) && (x < z)
```

evaluates to 1 if x is greater than y and less than z, and otherwise to 0. Another example:

```
(x == 7) || (y == 52)
```

evaluates to 1 if x is 7 or if y is 52, otherwise to 0.

Here's a more interesting example:

```
Object box "box" Physics_lab
 with name 'black' 'box',
  description "It's a box with a strange mechanism attached to it.";
  before [;
     Open: if (Schroedinger in Physics_lab || student in Physics_lab)
                "~Halt!~, screams a voice, ~You arre interrferring vith a most
                necessarry experriment!~";
  ],
  after [;
     Open: if (kitten in self && radiation_source has is_decayed)
                "There's a dead kitten inside the box.";
    ],
has static container openable;
```

The **before** routine intercepts the player's curiosity if *either* Erwin Schroedinger *or* his student are hanging around in the physics' laboratory. The **after** routine shows the macabre contents of the box if there is a kitten inside *and* if the radiation source has decayed (releasing the infamous poison gas).

4. Comparing Bitwise and Logical operators

The bitwise and logical operators are *similar* because:

- they use the same names: AND, OR and NOT.

- they use the same symbols: & | ~ for the bitwise operators, && || ~~ for the logical operators.

  However, they are *different* because:

- the bitwise operators treat each of the 16 (or 32) bits in a number individually (roughly like the arithmetic operators do), whereas the logical operators consider all 16 (or 32) bits as a single entity (roughly like the conditional operators do).

- the result of a bitwise operation can be any numeric value (for the Z-machine, -32768..0..32767), whereas the result of a logical operation can only ever be 0 or 1.

  The differences are important because, if you use the wrong operator, your game will still compile; worse, it will work correctly *some of the time*. For example, this statement:

```
if (x && y) print "CORRECT^";
```

  prints CORRECT if both x or y are **true** (that is, non-zero). This statement is probably wrong:

```
if (x & y) print "DUBIOUS^";
```

  because it will print DUBIOUS if the operation x&y evaluates to a non-zero quantity (the bit patterns of x and y must have one digit set to 1 in the same position). Here's another example:

```
if (~~x) print "CORRECT^";
```

print CORRECT if x is **false** (that is, zero). This statement is also probably wrong:

```
if (~x) print "DUBIOUS^";
```

because it will print DUBIOUS for *all values of x* apart from -1.

5. A "bit" of praxis

As we said, you are not likely to use bitwise operators unless you need to perform some low-level data manipulation or become concerned about wasting memory. Adam Cadre, author of many superb IF pieces, has published a library extension called Flags.h (it's part of a menu-based conversational system, used to manage what had been already said and when). You will find it here:

http://www.ifarchive.org/if-archive/infocom/compilers/inform6/library/contributions/

At the beginning of this file, he justifies its existence thus:

```
"This is a simple system to provide on/off flags that only take up one bit of memory; thus,
there's no need to waste memory by declaring a variable such as "doneflag" or some such,
allocating an entire eight bits to a variable that will never be anything other than 0 or 1."
```

You'll find a fine example of bitwise operator's usage in Flags.h, and although it's a surprisingly short piece of code, it's not easy to follow until you feel comfortable with some topics in Inform, such as arrays, routine calls and return values, and -- naturally -- bitwise operators.

On the other hand, logical operators abound in games. Each time you need to test for more than one condition going on (or definitely *not* going on) in a given scenario, a logical operator will elbow its way into your code with startling ease. Let's take a look at some common examples.

5.1 Using a logical AND

Suppose you want to do something (here denoted by printing YES) when x is 1 at the same time as y is 2. Here's one way:

```
if (x == 1)
  if (y == 2)
    print "YES^";
  else
    print "NO^";
else
  print "NO^";
```

Hard to read, error-prone, repetitious: ugh! Much much better like this:

```
if ((x == 1) && (y == 2))
    print "YES^";
else
    print "NO^";
```

5.2 Using a logical OR

Same sort of thing; you want to do something when x is 1 or when y is 2. The clunky way:

```
if (x == 1)
  print "YES^";
else
  if (y == 2)
    print "YES^";
  else
    print "NO^";
```

and the sexy way:

```
if ((x == 1) || (y == 2))
  print "YES^";
else
  print "NO^";
```

5.3 Using a logical NOT

You don't want to do anything *unless* x is 1 and y is 2

```
if ((x == 1) && (y == 2))
  else
print "YES^";
```

It somehow feels 'wrong' to need an **else** clause in these circumstances. Better to invert the condition:

```
if(~~((x == 1) && (y == 2)))
  print "YES^";
```

or alternatively to recast it thus:

```
if ((x ~= 1) || (y ~= 2))
  print "YES^";
```

(Note: ~~(A && B) is the same as (~~A) || (~~B), and ~~(A || B) is the same as (~~A) && (~~B).)

5.4 Avoiding run-time errors

One vitally useful technique for avoiding run-time errors relies on the logical operators working left-to-right. Suppose you wrote this as part of an object definition:

```
if (child(self) has general)
  print "DUBIOUS^";
```

to test if an object's eldest child object has its **general** attribute set. Unfortunately, you'll get a run-time error if the object happens to have no children. Here's how to save the day:

```
if (child(self) && child(self) has general)
  print "CORRECT^";
```

First check if there is a child; then *and only then* test its attribute. Here's a similar example for object properties. This may cause an error if object x doesn't define a **myprop** property:

```
if (x.myprop > 5)
  print "DUBIOUS^";
```

whereas this is safer:

```
if (x provides myprop && x.myprop > 5)
   print "CORRECT^";
```

5.5 Testing many different things

You can use conditions to check if an attribute has been set or not, what are the objects involved in an action, what action is being carried on, the current state of an object property, the value held in a variable... All these may be combined with logical operators as need arises. For instance:

```
if (action == ##AttackWith && noun == troll && second == axe) {
   remove axe;
   "The troll watches politely as the axe hits his head and breaks in a thousand
      pieces.  ~Zorry!~, he says, ~It zeemz that I didn't explain mezelf properly. It'z zwordz
      that do the trick.";
}
if (location == Fast_food_palace && (noun == burger || noun == fries)
       && clerk has friendly)
   "The clerk checks that nobody's looking and gives you a quick wink.
      ~I wouldn't recommend that unhealthy stuff, sir. Why don't you try our pizza
      instead?~";
if (noun provides size && noun.size >= 3 &&
       rolling_suitcase notin player &&
    player has exhausted) print_ret "You try to lift ", (the) noun, " but
    your arms seem to have other ideas.";
```

6. Conclusion.

As we have seen, bitwise and logical operators behave quite differently from each other even if their syntax is roughly similar. In game design, where it's necessary to constantly test the current state of objects and situations, logical operators become helpful tools to combine conditions, while bitwise operators may be safely ignored most of the time -- it never hurts, however, to know what they are and do, in case you come across a circumstance where they might prove needful.

[Thanks a lot to Roger Firth, whose unmerciful editorial pencil was decisive to bring

this article to its present form.]

Part E
Advanced Topics

Inform Extensions Voodoo

Introduction

n case you didn't get it from the title, this article describes techniques that are useful to Inform extension developers. These techniques offer solutions to especially sticky issues that extension authors face. They provide reusable code that not only makes life easier for the extension developer, but also eases the burden of implementing the extension for the game developer.

Note the distinction that is made here between "extension developer" and "game developer." Yes, I know that there probably isn't an extension developer that hasn't written, or at least started writing, a game. Keep in mind, though, that there is a whole set of issues that are faced uniquely by authors of reusable extensions. This article focuses on such issues, so readers should now take a moment to put on their "extension developer" hats to gain the full value of this article.

Be forewarned that the examples that follow are not for the light of heart. They implement code and techniques that the reader may find less than intuitive (well, maybe the first two aren't so bad). The rewards, however, far out weigh the learning curve associated with them. We'll start then, with the easiest and count down to the most difficult...

Trick #5: Replacing Replaced functions

The Replace keyword allows you to substitute your code for code that is defined in the standard library. This is a common way of extending the library, and for the game developer there is little need to think beyond this. The life of the extension developer is not as free as all that, however. Just as extensions tend to replace "portions" of the standard library in order to add new functionality, so will game designers often want to replace "portions" of extensions to special tailor what they do.

The problem? Because of the way system_file and replace have been implemented, this can only be done once. That is, when designated to be replaced, all instances of a function that follow the system_file directive (in a single file) are discarded, and only instances of the function that do not follow the system_file directive are included. That is, when an extension replaces a library routine (all of

which are defined in the context of system_file), it must define another version of that routine not under the system_file context. By definition, this new version cannot be replaced.

For a real world example, my ORNPCVerb.h module redefines several verbs and enables them to be performed by NPCs. Other than that modification, there are no changes to the verbs or the way they work. What if a game developer wanted to add new functionality to the TakeSub routine? Since he can no longer REPLACE this routine, is he forced to modify my library extension directly? Obviously this is a bad idea, but thankfully there is another solution, which is very easily done. Simply wrap the extension's new definition in conditional compilation directives:

```
Replace TakeSub;

#ifndef TakeSub;
  [TakeSub;
    !--Extension's new TakeSub code here
  ];
#endif;
```

This forces Inform to compile the extension's replaced version of TakeSub only if the game author hasn't already specified a version of his own.

Trick #4: Automatic Initialization

Sometimes the simplest modifications are the most powerful. This one is a biggie for extension developers. Although it is also useful to the game developer (indeed, I use it extensively when writing games), it is the extension author that gains an endless number of possible advantages:

```
Property additive object_initialise;
Replace Main;

[Main o;
  objectloop(o provides object_initialise && o.object_initialise~=0)
       o.object_initialise();
  InformLibrary.play();
];
```

Okay, I know what you're thinking: "That's it? That's the great key to writing library extensions? That's the Mustafa of useful features? You've got to be kidding!"

Trust me on this one. This small snippet, transparent if not used, is perhaps the single most useful modification that has ever been devised for the extension author. It enables a slew of functionality upon which additional extensions can be based, and allows for the initialization code of your extension to exist where it belongs: actually in your extension.

Keep an eye out, because this technique will pop up again in sections that follow.

Trick #3:Unified Entry points

A challenging area that extension developers face deals with the use of Library Entry Points (e.g.: UnknownVerb, ChooseObjects, ParseNoun). The problem is that each of these routines may only be defined once. Thus, if you write an extension that enables the use of a "wild-card for matching any collection of objects" (DM4 ex. 78) and another extension which "implements adjectives" (DM4 ex. 75), the two will conflict since both of these techniques define the ParseNoun entry point.

So how can you define two coexisting versions of the same routines? The short answer is that you can't. That is why game developers have always been forced to cut and paste the algorithms from two versions of an Entry Point routine into one and try to make sure that the two segments of combined code don't conflict.

It would be nice if there was such a thing as additive routines, just as there are additive properties. There isn't, but it is possible to simulate this idea by creating a list of routines and calling them until a specific value is returned (or not returned as is the case below). The following object is an implementation of this idea:

```
object AdditiveUnknownVerb
  with    routines 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  , num_routines 0
  , add_routine[rout;
     (self.&routines-->(self.num_routines))=rout;
     self.num_routines++;
   ]
  , run_until_not[untilnotval one two three four five six t retval;
     retval=untilnotval;
     for(t=0:t<self.num_routines:t++){
          if((self.&routines-->t)~=0)
               retval=((self.&routines-->t)(one, two,three,
                         four, five, six));
          if(retval~=untilnotval) return retval;
       }
       return retval;
     ]
;
```

It is now fairly straight forward to define a generic entry point routine that does

nothing but call our object for results (We'll use the UnknownVerb entry point):

```
[UnknownVerb;
  return AdditiveUnknownVerb.run_until_not(false);
];
```

Consider the above a mini-framework for extending UnknownVerb. Alone it accomplishes nothing in terms of UnknownVerb behavior, yet we now have gained the ability to allow multiple definitions of UnknownVerb to coexist and have them all execute in turn.

To exemplify this, we'll use two pre-existing versions of UnknownVerb (changing only the names). The first is based upon the Teleport.inf example, written by Michael Huang, which allows the player to teleport from one room to another by simply stating the name of the room:

```
[UVTeleport word place; !--UnknownVerb code
  objectloop (place has teleport){
    if(WordInProperty(word,place,name)){
       verb_wordnum=0;
       return 'teleport.room';
    }
  }
  rfalse;
];
```

Consider this definition to be part of extension one. The second definition of UnknownVerb is based upon code from the ORLibrary module ORMagic.h which allows magic spells to be cast by name, rather than forcing the player to use the "cast" verb. Consider this code to exist in extension two:

```
[UVCast word obj; !--UnknownVerb code
  objectloop(obj ofclass(ORMagic)){
     if(WordInProperty(word,place,name)&& obj.IsKnownBy(actor)){
       verb_wordnum=0;
       return 'cast';
     }
   }
   rfalse;
];
```

All that is needed to allow these two routines to coexist is to simply register them with our list. Although this traditionally would have been done by instructing the game developer to add certain lines to his initialise routine, the object_initialise

functionality which we discussed previously really shines here and eliminates the game author's requirements to do anything other than include your extensions:

```
!-- define in extension one
object with object_initialise[;
   AdditiveUnknownVerb.add_routine(UVTeleport);];

!-- define in extension two
object with object_initialise[;
   AdditiveUnknownVerb.add_routine(UVCast);];
```

And there you have it -- two implementations of UnknownVerb from two separate extensions working together instead of conflicting. (Why can't we all do that and just get along?)

Trick #2: Redefining existing objects

As we've already discussed, when modifying the standard library's behavior, extensions traditionally leverage Inform's REPLACE directive. This works well enough with stand alone routines, but there is a great deal of code in the standard library that is NOT contained in stand alone routines and so cannot be replaced with the "replace" directive. Instead, this code is contained in objects. The InformLibrary object is particularly noteworthy for this, and historically it has not been an easy task to write a library extension which modifies the code contained in the InformLibrary object. Instead, many authors resort to writing down instructions for their modifications to a standard library file, like "First change the lines starting at 638 of the parserm.h file which reads as ... to instead appear as ..." Not the easiest solution, but there is another way.

It is fairly easy to substitute an object's property definition with another definition at runtime:

```
[NewPlay; !--a modified version of InformLibrary.play()
   ...code goes here
];
[Initialise;
  InformLibrary.play=NewPlay; !--replace with new version
];
```

Of course an extension could leverage the object_initialise technique that we discussed above to eliminate the game designer from having to code anything in Initialise.

A more complete and somewhat more elegant form of this involves creation of an object that scans its own collection of properties and compares them to the properties of a destination object, replacing the ones with the same name. The following class does this:

```
class RedefineObject
  with dest_object 0
  , object_initialise[i count;
        count = #identifiers_table-->0;
        for(i=4:i<=count:i++){
           if(self provides i && self.dest_object provides i){
               (self.dest_object).i=self.i;
           }
        }
    ]
;
```

Using this class, an extension can redefine multiple properties of any object very easily. The following example will redefine both the begin_action and the end_turn_sequence routines of the standard library's InformLibrary object:

```
RedefineObject
  with dest_object InformLibrary !--destination object
  , end_turn_sequence[;
       !--new end turn code
    ]
  , begin_action[;
       !--new begin action code
     ]
;
```

It should be noted that unlike the REPLACE directive, the original version of the routines still exist since they are redefined at runtime. This is both a blessing and a curse, since it is possible to call the prior version of a routine under specific circumstances, but only at the cost of an increased code size.

And now... drum roll please...

Trick #1: Defining/Redefining Library Messages

What has seemed like a near impossibility faced by extension developers lies in the addition or modification of library messages. The standard library leverages the "before" property of the user-defined LibraryMessages object to enable the game developer to override the messages defined in the language definition file (usually English.h), but an extension does not have this luxury since the

LibraryMessages object can only be defined once. Not only does this keep a module from redefining the standard library's messages, it forces the extension developer to "hard code" messages into their objects. If the game developer wants to adjust the default responses he must modify the extension directly. What we really want is an additional level of messages. That is, a place where extensions can redefine or create library messages, which can then be redefined by the game developer if desired.

This can be accomplished by combining some of the techniques described above. First, let us make sure that a LibraryMessages object has been defined by the player and define one ourselves if they have not:

```
!--create only if not defined by game author
#ifndef LibraryMessages;
     object LibraryMessages with before[;rfalse;];
#endif;
```

Next we can create our "message proxy" which will provide storage for all of the new messages that are defined at the extension level. Using Trick #2, this object will also reprogram the LibraryMessages object to consider the registered extension messages (deferring to the messages defined by the game author first):

```
!--Object that maintains a list of Message routines (Tip #3)
object ExtLibMsgs
  with routines 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  , num_routines 0
  , add_routine[rout;
      (self.&routines-->(self.num_routines))=rout;
      self.num_routines++;
    ]
  , run_until [untilval one two three four five six t retval;
        retval=untilnotval;
        for(t=0:t<self.num_routines:t++){
              if((self.&routines-->t)~=0)
              retval=((self.&routines-->t)(one, two,three,
                           four, five, six));
              if(retval==untilval) return retval;
        }
        return retval;
     ]
  , routine_holder 0 !--to save the game author's Messages
  , object_initialise[; !--Auto setup Tip #4
        self.routine_holder=LibraryMessages.&before-->0;
           !--redefining the LibraryMessages object to call
           !--our routine instead (Tip #2)
        LibraryMessages.&before-->0=(self.DoMsg);
     ]
  , DoMsg[;
        if(ExtLibMsgs.routine_holder()~=true)
              return ExtLibMsgs.run_until(true);
        rtrue;
  ];
```

And now, a simple class will provide an interface for extension developers to leverage:

```
!-- The class definition used to redefine a message
class ExtMsg
  with object_initialise[;
          ExtLibMsgs.add_routine(self.body);
       ]
  , body 0;
```

With the above code in place, the extensions developer can now redefine library messages from multiple files in multiple places without conflicting with the game author's LibraryMessages Definition. For example:

```
ExtMsg with body[;
  Jump: if(location has hollow) "Loud echoes announce that
        you are jumping.";
  Wave: "You practice your break-dancing and do the wave.";
];
```

Note that this redefined message can be defined anywhere, unlike the InformLibrary object which must be defined specifically between Parser.h and VerbLib.h.

Conclusion

Okay. As I often say in my articles, the examples given above are just that - examples. They are intended to communicate the concepts, but several of the tips have been implemented better in the ORLibrary. If you are considering leveraging some of these ideas in your own extensions, it might behoove you to review some of the related ORLibrary modules. The code there is more fleshed out, have been written to support GLULX, and can serve as an example of real-world implementation.

Additionally, there are a couple of other "Tips" that I did not put in this article but which nevertheless target the same extension-developer audience. These can be found in the following articles:

Building a Personal library

And

Using "system_file" and "replace" In the Same Source File

Advanced NPC Creation: Introduction

he creation of a believable non-player character (NPC) is, at times, a daunting task. In part, this is because there is such breadth in the spectrum of detail that can be given to an NPC, but even this is just an element of the bigger truth: NPCs represent people, and people know people.

Unlike a book or a mirror, a person is alive. Living things by their very nature are not static. Even basic creatures, like a cat (no offense cat lovers!) walk around. They move with purpose, or perhaps they just wander aimlessly.

Living things also interact with the world around them. More than just seeking the warmth of a sunspot, cats knock things off counters and mercilessly attack helpless pairs of house shoes. They interact with other living creatures in different ways, such as rubbing against their legs or hoping into their laps. They also respond to interaction, like purring when pet.

People are even more complex, since they speak. Not only do they respond to questions, but they also initiate conversion. Language itself further adds to the complexity, since people learn. They can be told something, or read about it, and then relay that information to another person in conversation.

To some degree a player will expect these things from an NPC, too. Players of interactive fiction, new or experienced, experiment with the world in which they are playing. The player knows what happens in real life and tries actions to see if they work in the game.

Players will invariably test to see what the NPCs are capable of. They understand intuitively that people are subjective and that they may choose to either do something or refuse to do something simply because they are in a good or bad mood. When dealing with people who refuse them, players will often attempt persuasion by complimenting, or trying to establishing a camaraderie with a character. They may even attempt bribery.

What the NPC is capable of will add to its believability. The NPC's believability will add to the player's enjoyment and therefore overall assessment of the game.

So what's the big deal?

A believable NPC is just not an easy thing to accomplish.

This multi-part article series will describe some of the hurdles that are encountered when developing NPCs, describe design principals that can be used to address these hurdles, and give INFORM specific examples of implementing the principals. The examples vary in complexity and are implemented with intermediate to advanced coding techniques. These techniques include:

- creating new base classes to inherit from
- creating new print rules
- extending the standard library
- replacing the language definition file

The results are more than worth the effort, however, and include dynamic text that reads more naturally, library messages that relate to NPC actions, and a more believable, more powerful NPC implementation.

For more complete examples of these principals in action, see the ORNPC and ORKnowledgeTopic entries in the ORLibrary section of this site.

(Note: at the time if this writing, the ORKnowledge and ORNPC entries are in ALPHA state. That is, they are still actively being developed and bugs are being ironed out, however they are developed enough to sufficiently serve as example code that will compile and run.)

Advanced NPCs, part 1: Pronouns on Steroids

Common Print Rules

ometimes writing reusable code is a hassle. This is especially true in the case of pronouns. When creating a simple NPC, one may be tempted to hard code gender phrasing into text. Sentences like

```
The waiter gives you his pen.
```

or

```
The doctor drops her stethoscope.
```

may very likely find their way into such an object. Although this seems fine at first glance, and indeed IS fine for instances of an object, text written in this way ties the object to its gender. Dropping "HER stethoscope" is fine for a female doctor, but a male doctor should drop "HIS stethoscope." This can restrict an object's reusability and therefore makes it unsuitable for implementation in a class.
The Inform Developer's Manual (fourth edition) gives a limited example of how to deal with this in chapter IV §26 exercise 62 by defining print rules to deal with a couple of different forms of pronouns (nominative and accusative.)

So everyone will be on the same page, lets review one of the examples given in §a6 page 477:

```
[PronounNom i; !this code snatched from the DM4
  if (i hasnt animate || i has neuter) print "it";
     else {
        if (i has female) print "she";
        else print "he";
     }
];
```

This routine, and the others like it, simply checks for the existence of the appropriate gender attributes and prints out the correct pronoun (i.e.: "him," "her," or "it.") The example in the manual is, I think, intended to point us in the right direction without really addressing all possibilities. That's for us to do.

Why Are There So Many Pronouns?

Some readers may not immediately recognize the references to "Nominative" or "Accusative" forms of a noun or pronoun. Many people, way back in grammar school, learned instead to call these "subject of a sentence" and "object of a

sentence." Once upon a time, in languages other than English, all nouns had different forms based upon their placement in the sentence. Ancient Greek, for instance, has four (actually five, but only four of significance). These are:

- Nominative (subject of the sentence)
- Accusative (object of the sentence)
- Genitive (noun possessing or qualifying another noun)
- dative (receiver of the object)

These correlate exactly with how nouns are used in a sentence. For example:

```
"The king          : nominative (subject of sentence)
put the wife       : accusative (object of sentence, what was put)
of the banker      : genitive (qualifier of the wife)
in the dungeon."   : dative (receiver of the object)
```

And again:

```
"The boy           : nominative (subject of sentence)
kicked the girl"s  : genitive (owner of the ball)
ball               : accusative (object of sentence, what was kicked)
into the creek."   : dative (receiver of the object)
```

In English, differences between these forms have primarily disappeared. An apostrophe "s" has been adapted to show possession, and most nouns are now written the same regardless of their placement in the sentence. With pronouns, however, they remain in varying degrees. The dative form has all but been eliminated and shows up only when nouns affect themselves, but a second genitive form also occurs when the qualified noun has been previously described and is so obvious as to be left out.
See how the masculine pronoun takes a different form depending upon its placement:

```
HE gave HIM HIS dog, which was HIS, anyway.
```

The last occurrence of the masculine pronoun ("his") is the second form of

possession. In this example the two forms are the same. This is not the case for the feminine pronoun:

```
SHE gave HER HER dog, which was HERS, anyway.
```

The second form of possessive pronouns occurs when the object possessed is obvious and therefore not listed. For example:

```
She took her chocolate, but gave you YOURS.
```

does not specify chocolate in the same dependant clause as the pronoun "YOURS". If it did, then the form of the pronoun would revert to the first possessive form and the sentence would read:

```
She took her chocolate, but gave you YOUR chocolate.
```

A Plethora of Pronouns

The following table shows the pronoun for each person, plurality, and gender:

Person/Plurality/Gender	Nom.	Accus.	Gen. 1	Gen. 2	Dat.
1P/Sing	I	me	my	mine	myself
1P/Plur	we	us	our	ours	ourselves
2P/Sing*	you	you	your	yours	yourself
2P/Plur*	you	you	your	yours	yourselves
3P/Sing/Masc	he	him	his	his	himself
3P/Sing/Fem	she	her	her	hers	herself
3P/Sing/Neut	it	it	its	its	itself
3P/Plur	they	them	their	theirs	their selves

** As a side note, notice that in modern English, second person pronouns almost always are treated as plural even if they represent a singular noun. This was not always the case. For a game placed during the reign of King James, it may behoove the developer to implement the missing second-person, singular pronouns that were commonplace in old-English: Thou, Thee, Thy, Thine, and Thyself, respectively.*

Looking at this table, you may be wondering what some of these forms have to do with writing an NPC class. Specifically you may wonder about first, and second person pronouns as well as the plural pronouns. The answer to this is that you

simply don't know how your class may be used in the future. For example, the standard library supports the ability to allow the player to become an NPC (the ChangePlayer() library routine). It may be that phrases like "The troll swings the axe" need to become "You swing the ax."
Additionally, some library extensions (such as the OREnglish.h file) also allow the game to change the "person" the narrative is being told in. The story could be told in first person and the above example would then need to read, "I swing the axe."

Plurality too is not beyond reason since an NPC can be constructed to represent a swarm of bees ("They fly after the horse as it runs from them") just as it could represent Superman ("He flies after the horse as it runs from him.") A game where the player shares a body with an NPC might require first person plurality. Text would suddenly need to take the form "We swing the ax" and "We fly after the horse as it runs from us."

When developing reusable code, it's always best to cover all bases.

Enough with design, now the implementation.

A Stronger, More Powerful Pronoun Rule

Using the above chart, we"ll rewrite the PronounNom routine and name it "I" after the first person singular pronoun (remembering the pronoun is a lot easier than remembering the name of the case of the pronoun, and first person singular is the only form that offers five distinct words for each case):

```
Default NarativePerson 2; !if not changing the
  !narrative, then default
  !to second person
[I obj;!Nominative form (she/he/it/they/I/we/you)
  if(obj==player){  !talking about the player?
     if(NarativePerson==1) { !first person?
        if(obj has pluralname) print "we";
        else print "I";
     }
     else print "you"; !2nd person, ignore plural
  }
  else { !not the player, so 3rd person
     if(obj has pluralname) print "they";
     else { ! is singular
        if (obj has neuter || obj hasnt animate)
           print "it";
        else {
           if (obj has female)  print "she";
           else print "he";
        }
     }
  }
];
```

In addition to the above print rule named "I", a rule for each of the five cases should be implemented, plus additional capitalized versions. By duplicating the above example and modifying it just a little, we will end up with ten basic pronoun print routines. Specifically, the five lowercase routines: "I", "Me", "My", "Mine", "Myself", plus the five uppercase version: "CI", "CMe", "CMy", "CMine", and "CMyself".
Using these print rules, it is now possible to write lines of text that literally transform themselves based upon their use. Look at the following example for a moment:

```
print (CI)man, " gave ", (My)man," food to ",
  (Me)woman,". ", (CI)woman," accepted ",
  (My)man, " food and then thanked ",(Me)man;
```

With two NPCs, one male, one female, this line above would output the following text:

```
He gave his food to her. She accepted his food and then thanked him.
```

If the player inhabits the object "man" then the text would appear as:

```
You gave your food to her. She accepted your food and then thanked you.
```

Or if the NarativePerson variable is equal to 1 (first person):

```
I gave my food to her. She accepted my food and then thanked me.
```

Or the player could be the woman and the man object could represent a group of monks:

```
They gave their food to me. I accepted their food and then thanked them.
```

The variations are numerous.

Even More Flexibility...

The whole purpose in having pronouns is to keep from having to repeatedly refer to the noun by name. For example, a sentence such as:

```
The troll reached into the troll's bag and pulled out the troll's axe. The troll
swung the troll's axe and chopped down the troll's tree.
```

reads better as:

```
The troll reached into his bag and pulled out his axe. He swung it and chopped down
his tree.
```

Once the noun has been mentioned, it is fine to begin referring to it in pronoun

form. It is even desirable to do so. Using the pronoun without first naming the noun, however, is confusing. For independent blocks of text, that are displayed solely as we write them, this is never a problem since we can be assured that we reference the troll first by name, then as a pronoun. NPCs, however, have a tendency to do multiple things at different times, and it is not always known what text has already been printed.

Consider an NPC that moves around, possibly responding to a call, and speaks 50 percent of the time. Creating text that describes the NPC entering the room and another line of text that describes him talking makes it difficult to efficiently use pronouns because one of the events may happen without the other, or they may both happen together. Assuming both occur, we would like to print the following:

```
The troll enters the room. "You called? " he asks.
```

If we know that these two sentences will always print together, we can code them by referring to the troll with the standard "The" print rule in the first line of text, and our specialized "I" print rule in the second line of text. This doesn't really work well when the NPC talks without first moving, however, since we would get a pronoun without any prior mention of the troll:

```
"You called?" he asks.
```

This leaves it up to the player to figure out who the pronoun "he" refers to. To complicate things further, if the troll is mentioned prior to his entering the room, we get duplicated references to the noun. Although this is okay:

```
You blow the whistle to summon the troll. The troll enters the room.
```

The pronoun is preferable:

```
You blow the whistle to summon the troll. He enters the room.
```

In addition, when two objects are of the same gender, pronoun use can become confusing:

```
The barber grimaced at the sheriff. He handed him a badge.
```

Who gave whom a badge? The sentence seems to indicate that the barber gave the sheriff the badge, but was that really what we intended? In cases such as these, it is generally preferable to limit pronoun use to only the first named object of any given gender:

```
The barber grimaced at the sheriff. The sheriff handed him a badge.
```

Preceding text should also change the pronoun context:

```
The sheriff entered. The barber grimaced at him. He gave the barber a badge.
```

Addressing both of these points requires another set of print rules that keeps

track of the current pronoun-described objects and outputs the name of the object if the use of a pronoun would be unclear:

```
Global MPronounObj;
Global FPronounObj;
Global NPronounObj;
[TheI obj checkobj;
  if(obj hasnt animate || obj has neuter)
     checkobj=NPronounObj;
  else {
     if(obj has female)
        checkobj=FPronounObj;
     else
        checkobj=MPronounObj;
  }
  if(obj==checkobj)  print_ret (I)obj;
  print (the)obj;
  if(checkobj==NULL && obj~=player){
     if(obj hasnt animate || obj has neuter)
        NPronounObj=obj;
     else {
        if(obj has female) FPronounObj=obj;
        else MPronounObj=obj;
     }
  }
];
```

Of course nine more rules need to be implemented to cover the rest of the print rules developed previously. Additionally, a routine should be created that resets pronoun usage by setting the three pronoun variables to NULL:

```
[ClearPronoun;
  MPronounObj=FPronounObj=NPronounObj=NULL;];
```

Now the above example can be demonstrated by the following randomized code (assuming the sheriff and barber objects have been defined appropriately):

```
ClearPronoun();
if(random(2)==1) print (CTheI) sheriff," entered. ";
if(random(2)==1) print (CTheI) barber," grimaced
  at ",(theMe)sheriff,". ";
print (CTheI)sheriff," gave ",(theMe)barber,"
  a badge.";
```

Verb and (pro)Noun agreement...

The reader may have noticed that many of the examples above have been given in past tense. This is not only to demonstrate diversity, but also for the sake of clarity, since it avoids a quirk in the English language that should be discussed now. Specifically it is the agreement of verbs with their Nominative noun's

plurality when described in third person, present tense.

Yeah, I know, it sounds confusing, but English speakers understand this on an intuitive level so a couple of examples should clear this right up. First lets look at a present tense verb used with a first person noun:

```
I run.

We run.
```

Okay, that was both singular and plural forms. No change in the verb. We can see the same thing in second person:

```
You run.
```

Even third person plural follows this rule:

```
They run.
```

The third person, singular noun disrupts our continuity, however, because it makes the verb change form:

```
He runs.

She runs.

It runs.
```

For most verbs, this change takes the form of an "s" being appended to the verb. Not all, however:

```
We do.

He does.
```

and,

```
They fly.

It flies.
```

And again,

```
I wax.

It waxes.
```

Print rules won't help us here. What we need is a routine that checks the noun and chooses the appropriate verb form, or possibly appends an "s" to the verb for us if appropriate. Luckily, verbs only follow nominative nouns so a single lowercase and a single uppercase routine can be put into place to affect this change. Here's the uppercase version:

```
[CIVerb obj verb altverb;
  print (CTheI) obj," ";
  if(obj has pluralname || obj~=player)
     print verb;
  if(altverb==null)
```

```
      print verb,"s";
  else print altverb;
];
```

Routines always return a value of some kind. Roger Firth's article, "Print Rules" suggests creating a do-nothing print rule to discard the return value rather than print it:

```
[ignore o;];
```

Having done this, the IVerb routines can be implemented in the print statement like this:

```
print (ignore)CIVerb(obj,"drink"),"
  the glass of milk.";
```

And like this:

```
print (ignore)CIVerb(obj,"smash",
  "smashes"),"  the empty glass.";
```

Depending upon the attributes of obj and what text has already been printed, any of the following could be printed:

```
The troll drinks the glass of milk. He smashes the empty glass.

You drink the glass of milk. You smash the empty glass.

The monks drink the glass of milk. They smash the empty glass.
```

As If That Wasn't Enough...

Additionally, there are a few special cases regarding the so-called "verbs of being" that change based upon plurality and person. The standard library attempts to address these with the IsOrAre print rule. If implementing a game in first person, this routine would have to be replaced to also support "am". Better still, a variable could be implemented that checks the tense a game is being told in and adequately supports "was", "were", and "will".

A Little Note on the Code

The samples described above are just that, samples. Although the text often refers to duplicating code and changing portions of it to match, this really isn't the best way. A skeleton function containing the duplicated algorithm which accepts all variations of text as parameters would take up significantly less space, and be easier to maintain. You can see for yourself because...

The ORLibrary entry ORPronoun.h contains a complete implementation of the techniques discussed in this article.

Knowledge and Conversation
§1 The Basics
(Advanced NPC Implementation, Part 2a)

 s developers, we can put forth a great deal of effort while trying to make an NPC appear believable. We can make the character walk around randomly. We can make him steal items that are lying around. We can even program the character to scribble graffiti on walls at significant locations. It's true, a character can be designed to do just about anything imaginable, but there is one aspect of NPC design that will, above all others, make or break the believability of an NPC: conversation.

A successful implementation of a PC-NPC conversation is no small task. Many developers have tried and failed to create believable dialog. Obviously, success at this bares great dependency upon the author's writing skill. Conversely, a well-written manuscript can be implemented poorly. The truth is, a text adventure without code is just text. Code and prose are equally important in interactive fiction and the two must complement each other to truly be effective.

The purpose of this three-part article series is to identify the major coding techniques used in the development of talking NPCs and give examples of implementing some of these. Additionally we will cover advanced techniques such as uniting knowledge and information, separating information from conversation, morphing conversation topics, conversation scripts, and implementing knowledge that is learnable. First, a disclaimer:

In interactive fiction, complex puzzles often have multiple solutions. The similarities between life and art are ever obvious in this regard, since complex coding objectives often can be accomplished by multiple techniques. As a general rule of thumb, the more complex the objective, the more diversified are the possible solutions. Since this is an article about "advanced" techniques in NPC conversations, the variations and alternate implementations abound. It should be noted that the techniques covered in this article work for me, but my way is not the only way. If you, as the reader, see another technique then feel free to implement it.

Now that that stuff is out of the way, let us begin slowly by touching upon the three major techniques used to implement interactive character conversations: Menus, keywords, and topics.

The Menu System

A commonly used method of implementing conversation is the menu-based system. There are several advantages to using this technique:

- There is never any doubt for the player that there is a conversation to be had (i.e.: that there are things to say or ask) since the conversation options are listed up front.
- The player never has a problem with phrasing (finding the right words to ask).
- The developer has control of the flow of conversation. Specifically, the author determines what can be asked and when that menu option appears.

There are also some disadvantages to this technique:

- The player's experience is a less personal one. Choosing conversation options is more akin to the "Choose Your Own Adventure" style of stories. It can also be frustrating to a player that wants to ask a question which isn't listed. Given the same conversation coded two ways, many players come away feeling that their options seemed more limited with this technique. A player that has the ability to formulate his own question experiences a feeling of freedom, even if the question has not been coded for and the response is just the simple: "I know nothing about that."
- The menu technique also requires less player involvement. In conventional methods, important conversation topics can be derived from seemingly insignificant details mentioned previously. For example, asking about a book of restaurant matches, found in a character's coat, might reveal him as a previously unknown witness to a restaurant murder. In a mystery game, the matches might be a clue that the player would need to attend to and recognize the significance of. Seeing the menu item "Ask about matchbook" fairly spoils the need for the player to pay attention to this sort of detail.
- A significant drawback to menu-based conversations is the inherent limitation of realism. A believable NPC is often coded with dozens of insignificant responses that have no importance in the game except to add realism to the character. Implementing this sort of detail in a menu-based system is difficult at best. If not carefully limited in scope, the player could be left facing a substantially sized menu of meaningless small talk.

This is not to say that the menu system is not worth considering. In fact, it has been implemented with great success in a few choice works. It is, however a replacement for what the standard library offers, the Ask/Tell/Answer paradigm, and will not be discussed further in this article.

Nearly every other conversation technique is based upon the Ask/Tell/Answer model. At its most basic implementation is the use of keywords...

Using Ask/Tell/Answer (With Keywords)

As stated above, the standard library implements a basic conversation framework by defining three communication verbs. These three verb are 'ask,' 'tell,' and 'answer' (or say) and they are handled traditionally in the NPC's life property. For the 'tell' and 'ask' forms, the first significant keyword is stored in the library variable 'second.' For the 'answer' form, it is stored in the variable 'noun.' Take the following as an example:

```
object -> magicmirror "magic mirror"
has animate
with name 'magic' 'mirror'
,   life [;
        tell: if(second=='hello')"~Yes, I know what 'hello' is.
           I may not get out much, but I recognize a greeting!~";
        answer: if(noun=='hello') "~Well, 'hello' right back at you,
           mortal.~";
        ask: if(second=='hello') "~You want to know about 'hello?'
           Where have you been all your life, that you don't
           recognize a greeting?~";
     ];
```

See how the keyword 'hello' is handled differently depending upon the verb used to reference it:

```
>ask mirror about hello
"You want to know about 'hello?'  Where have you been all your life,
that you don't recognize a greeting?"

>tell mirror about hello
"Yes, I know what 'hello' is. I may not get out much, but I recognize a
greeting!"

>say hello to mirror
"Well, 'hello' right back at you, mortal."
```

Alone, this technique is fairly limited, but it is also one of the most widely used methods for implementing conversation that you will find in Inform games.

As such, there are various methods of making it somewhat more powerful. The consult_from and consult_words variables, for example, can be used to "look ahead" and example more than a single word. To demonstrate a starting point for diversifying the keyword technique, let's focus on the ASK form of communication in the next example:

```
object -> magicmirror "magic mirror"
  has animate
  with name 'magic' 'mirror'
  , life [word;
        ask:wn=consult_from; !position the pointer for the
           word=NextWord(); !NextWord call
           if(word=='hello') { !make sure we are talking about a type of hello
              while((word=NextWord())=='in' or 'from'); !ignored words
              switch(word) { !lets find what kind of hello we are looking for
                 0: "~You want to know about 'hello?' Where have you been
                    all your life, that you don't recognize a  greeting?~";
                 'canada', 'canadian': "~In Canada they say 'Hello, eh.'~";
                 'hawaii', 'hawaiian': "~Hmmm, I believe that would be
                    'Aloha.'~";
              }
           }
     ]
;
```

Now the keyword 'hello' can be put into a specific context:

```
>ask mirror about hello in canada
"In Canada they say 'Hello, eh.'"

>ask mirror about hello in hawaiian
"Hmmm, I believe that would be 'Aloha.'"

>ask mirror about hello
"You want to know about 'hello?' Where have you been all your life, that
you don't recognize a greeting?"
```

Obviously the above example isn't as useful as it could be since it is too dependent upon word order. For instance,

```
>ask mirror about Hawaiian hello
```

will NOT be matched by the routine in its present form. Certainly it could be modified to scan for words despite order, but the more we pattern match, the more we are duplicating code that exists in the parser. For conversation topics that are more than a single unique keyword, there is a better way.

Using Ask/Tell/Answer (With Conversation Objects)

Inform is an object based language. The rooms the player visits are objects, the items that are gathered are objects, and the very NPCs that exist in the game are objects. Not surprisingly, all of these things are accessible through the game. We can look at the room we are in, pick up a hammer that lies on the floor, and talk to the NPCs. Each of these objects exists in the game world. One of the more allusive concepts for a new Inform developer to grasp is that objects do not have to actually be present in the game at all. They can, instead, be abstractions. Use of these non-existent objects is the third, and most powerful method of implementing NPC conversation, since it opens the door for numerous other techniques.

Let us set the stage for more advanced techniques by walking through a thorough example of basic conversation objects. First, we can create a base class from which all conversation objects will be derived:

```
class KnowledgeTopic;
```

Not the most impressive line of code, but it is a step in the right direction. In fact, what it accomplishes is significant. We can now, for instance, write a fully qualified scope rule to determine if an NPC "owns" a specific conversation object and extend the 'ask' verb to use that rule.

```
Object CommonKnowledge;
[TopicInTarget o; !isolate information known by the target
  switch(scope_stage){
     2:objectloop(o ofclass KnowledgeTopic &&
        parent(o)==inputobjs-->2 or CommonKnowledge)
        PlaceInScope(o); rtrue;
     3: "I did not understand who or what you were referring to.";
  }
];
extend "ask" first * creature 'about' scope=TopicInTarget-> Ask;
```

The TopicInTarget() routine (scope rule) will pull into scope any KnowledgeTopic derived object that is a child of the character being questioned. In this way we can define what the character "knows". Additionally, notice the definition of the CommonKnowledge object. Topics that are placed within this object are also pulled into scope. All characters know topics that are classified as "common knowledge."

Here is an example of two topics. The first, "beauty," is a child of the CommonKnowledge object and so is known by all characters. Only the mirror knows the second, "snowwhite":

```
KnowledgeTopic beauty CommonKnowledge with name 'beauty';
KnowledgeTopic snowwhite magicmirror with name 'snowwhite' 'girl';
```

As you can see, using knowledge objects allows us to fill the 'name' property with the keywords that can reference a topic, just as we would a table or a chair. With this technique, the variable 'second' no longer holds the keyword, but points to the object that was recognized. The life property must be changed to reflect this:

```
object -> magicmirror "magic mirror"
  has animate
  with name 'magic' 'mirror'
  , life [; ask:switch(second) {
        beauty: "~Ah yes. That is only skin deep.~";
        snowwhite: "~Yes, I remember her.  But that was long ago.~"; }]
;
```

For the sake of example, and to verify that our new scope rule is working how we intended it, let's create another character coded almost exactly as the mirror:

```
object -> guard "guard"
  has animate
  with name 'guard'
  , life [; ask:switch(second) {
        beauty: "~Ah yes. That is only skin deep.~";
        snowwhite: "~Yes, I remember her.  But that was long ago.~"; }]
;
```

Now we have two characters to converse with. Note that, like the mirror, the guard is also coded to respond to the 'snowwhite' question. It'll be up to our scope rule to determine if he knows about the topic. Since he does not, this portion of his life rule will never be called and the text, although defined, will never print:

```
>ask guard about beauty
"Ah yes. That is only skin deep."

>ask mirror about beauty
"Ah yes. That is only skin deep."
```

```
>ask mirror about snowwhite
"Yes, I remember her. But that was long ago."

>ask guard about snowwhite
There is no reply.
```

That was what we expected. The guard knows nothing of 'snowwhite.' Our new scope rule is pulling in the appropriate knowledge for the appropriate character.

On to Advanced Techniques (Concluding the Basics)

The topics discussed in this section have set the stage for a discussion of more advanced techniques. In §2 we will begin to delve into more advanced topics that build upon what we have learned so far. For readers that wish to further contemplate the material of this section, I highly recommend a review of Roger Firth's Infact page. Specifically, the conversation section found at " http://www.firthworks.com/roger/infact/convers1.html" is of immediate relevance.

And now, on to section 2...

Knowledge and Conversation
§2 Advanced Techniques
(Advanced NPC Implementation, Part 2b)

n the previous section we discussed various methods for developing NPC conversation and reviewed techniques for implementing the most widely used of these. In this section we will expand upon those methods with much more powerful techniques and begin to develop reusable classes for NPCs in the process.

Uniting Knowledge and Information

In the previous example, a character's knowledge of something and his ability to talk about it are not as tightly related, as they ought to be. Coded into the guard's life routine is the same information about Snow White that the mirror has. The guard may not recognize her name, but he was coded, wastefully, to talk about her anyway. The reverse of this is a more probable circumstance, considering the existence of common knowledge. In the above example, a new NPC would also need to be coded with responses to the 'beauty' topic. Uniting what is said with what is known is the second major benefit to using conversation objects.

To start off with, let's modify our KnowledgeTopic base class a little to default a property, which we will arbitrarily call "TopicInformation:"

```
class KnowledgeTopic
  with TopicInformation "~Yes, I know about that but I think I'll keep it
     to myself.~"
;
```

Now we can tie the text to the actual knowledge objects:

```
KnowledgeTopic beauty CommonKnowledge with name 'beauty'
  , TopicInformation "~Ah yes. That is only skin deep.~"
;

KnowledgeTopic snowwhite magicmirror with name 'snowwhite' 'girl'
  , TopicInformation "~Yes, I remember her.  But that was long ago.~"
;
```

And remove the text from the life property (of both characters) entirely. Instead, we can print out the TopicInformation value:

```
object -> magicmirror "magic mirror"
  has animate
  with name 'magic' 'mirror'
  , life [;ask:
     if(second ofclass KnowledgeTopic)  {
        PrintOrRun(second, TopicInformation);
        return true;
     }
];
object -> guard"guard"
  has animate
  with name 'guard'
  , life [;ask:
     if(second ofclass KnowledgeTopic)  {
        PrintOrRun(second, TopicInformation);
           return true;
     }
];
```

Note that the example runs exactly as it did previously (i.e.: The guard will respond when asked about 'beauty', but not 'snowwhite.' The magic mirror will talk about both.)

An NPC'a ability to converse about a topic is now tied directly to the scope of the topic itself. If the character "knows" the information, he can talk about it. No changes to his life property need to be made to add this knowledge. This technique benefits us in several ways:

- The life property now has code that is consistent across all NPCs. This enables us to implement a NPC base class to share the common code in the life routine.
- We now have the ability to add knowledge and to take away knowledge without coding them in each and every NPC. We simply have to move the knowledge object around.
- We are now one step closer to implementing a feature in our NPCs that is somewhat uncommon: the ability to learn and teach. Of course, this will require a few changes to fully accomplish. The next step in that direction is...

Shared Knowledge

So far we've developed our scope rule to allow a knowledge object to be known by one character or all characters. This doesn't cover all cases, though. Rarely do we have an all-or-nothing circumstance; often we have a group of people that know

a specific topic of conversation, and another group of people that do not. What is needed is a way to keep track of who knows what.

The most intuitive way to accomplish this would be to add a property to each NPC that records what he knows. This isn't really the best solution, however, since knowledge does not have to be limited to NPCs. It could also be found in a book. Instead we can attach a property list to the knowledge object that indicates who or what "knows" it:

```
...from KnowledgeTopic class
  , KnownBy 0 0 0 0 0 0 0 0 0 0 !added to KnowledgeTopic
```

Having done that, it is now not such a complicated task to check this list for a specific object. We'll call this routine IsKnownBy() and attach it as a property to the KnowledgeTopic class as shown below:

```
class KnowledgeTopic
  with TopicInformation "~Yes, I know about that but I think I'll keep
     it to myself.~"
  , KnownBy 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  , IsKnownBy[obj t;
     for(t=0:t<(self.# KnownBy)/2:t++) (
        if(self.& KnownBy -->t==obj) return true;
     }
     return false;
  ]
;
```

And, of course, our scope rule can leverage this as well:

```
[TopicInTarget o; !isolate information known by the target
  switch(scope_stage){
     2: objectloop(o ofclass KnowledgeTopic)
        if(parent(o)==CommonKnowledge ||
              o.IsKnownBy(inputobjs-->2))
           PlaceInScope(o);
        rtrue;
     3: "I did not understand who or what you were referring to.";
  }
];
```

Note that our scope rule no longer looks for the parent of the topic anymore. This is actually a preferable arrangement since we can now avoid placing the knowledge topics in the game world at all. This keeps us from having to deal with sticky

commands like:

```
>drop beauty
```

or

```
>examine beauty
```

To demonstrate our new capacity for selectively shared knowledge, we will need to create a third NPC. This one, like the mirror will also know the 'snowwhite' topic, making this knowledge shared by two characters, but still unknown by the third (the guard.)

Since we are beginning to duplicate code, this is a good opportunity to implement an NPC base class:

```
Class NPC
 has animate
 with life[;ask:
       if(second ofclass KnowledgeTopic) {
          if(noun==actor) {
             L__M(##Tell, 1);
             rtrue;
          }
          PrintOrRun(second, TopicInformation);
          rtrue;
       }
    ]
;
```

The two existing NPC's, by inheriting from this class, are now reduced to:

```
NPC -> magicmirror "magic mirror" with name 'magic' 'mirror';
NPC -> guard "guard" with name 'guard';
```

And a third can be created just as easily:

```
NPC -> dwarf "dwarf" with name 'dwarf';
```

The snowwhite topic can now be shared between the MagicMirror NPC and the

Dwarf NPC by filling in values in the KnownBy property (the beauty topic, being a child of the CommonKnowledge object, does not need the KnownBy property):

```
KnowledgeTopic snowwhite magicmirror
  with name 'snowwhite' 'girl'
  , TopicInformation "~Yes, I remember her.  But that was long ago.~"
  , KnownBy magicmirror dwarf 0 0 0 0 0 0 0
;
```

Now we can verify that the same KnowledgeTopic is shared by two NPCs, but not all NPCs:

```
>ask mirror about snowwhite
"Yes, I remember her. But that was long ago."

>ask guard about snowwhite
"There is no reply."

>ask dwarf about snowwhite
"Yes, I remember her. But that was long ago."
```

Now we're are ready to give our NPCs (and PCs, too) the ability to learn...

Learning and Teaching Knowledge

Learning and teaching are the same action described from two perspectives. Since implementing the 'shared knowledge' functionality, the act of learning a new topic translates exactly into manipulating the topic's KnownBy list. The following routine can be added to the KnowledgeTopic class to do just this:

```
...from KnowledgeTopic class
  , MemorizeFor[obj t;
     if(self.IsKnownBy(obj)) return true; !already known
     for(t=0:t<(self.# KnownBy)/2:t++) {
        if(self.& KnownBy -->t==0){
           self.& KnownBy -->t=obj;
           return true;
        }
     }
     print "Run-Time Error!  Object is known by too many sources and
        cannot be learned.  Increase the size of the KnownBy property.";
     return false;
  ]
```

Now, letting the guard in on the 'snowwhite' secret can be accomplished with a simple:

```
snowwhite.MemorizeFor(guard);
```

Thus far, the 'ask' portion of the 'life' routine has called the TopicInformation property to print out the topic. Often, as with the case of topics being taught and learned, there is more to be done than simply printing text. Morphing topics is an example of this that will be discussed in a moment, as well as adding character personality by separating information from conversation. For now though, we can just recognize that the teaching/learning of topics should be tied to the printing of dialog and a driving function should be added to call both, without blurring the two into the same routine. A new property, we'll call it 'TellAbout,' will be added to the KnowledgeTopic class to do both:

```
Attribute learnable;

... rest of KnowledgeTopic class code here
  , TellAbout[to from;
     PrintOrRun(self,TopicInformation);
     if(self has learnable) Self.MemorizeFor(to);
  ]
```

The TellAbout routine is where the logic behind the conversation occurs. The 'learnable' attribute has been introduced to govern whether or not to "teach" this topic. Some topics are simply small talk and don't need to be shared between characters. 'Learnable' allows the developer to designate which topics can be "learned."

Astute readers may catch that the 'from' variable is never used and so will generate a compiler warning. This is okay. The 'from' variable is actually a parameter that will become useful to us later. We've just introduced it early.

We can now change the life routine of our NPC class to call the TellAbout property:

```
...from NPC class
  , life [;ask:
     if(second ofclass KnowledgeTopic) {
        second.TellAbout(actor,self);
        return true;
     }
  ]
```

That's all that's needed for the player character to learn from an NPC. Of course, learning a topic is nothing if you cannot speak about that topic. In order to do that, we need to extend the 'tell' verb. First, a new scope rule to check and see what the actor (usually the player) knows:

```
[TopicInActor o; !isolate information known by the actor
  switch(scope_stage) {
     2: objectloop(o ofclass KnowledgeTopic &&
              (o.isknownby(actor) ||
              parent(o)==CommonKnowledge))
           PlaceInScope(o);
        return true;
     3: "Who did you want me to say that to?";
  }
];
```

Next we will extend the 'tell' verb appropriately:

```
[TellTopicSub;
  if(noun==actor) {
     L__M(##Tell, 1);
     return;
  }
  second.TellAbout(noun,actor);
];
Extend "tell" first  * creature 'about' scope=TopicInActor -> TellTopic;
```

And that's it. All the functionality for teaching and learning topics is in place. For the sake of example, let's give the 'snowwhite' topic the 'learnable' attribute:

```
KnowledgeTopic snowwhite magicmirror
  has learnable
  with name 'snowwhite' 'girl'
  , TopicInformation "~Yes, I remember her.  But that was long ago.~"
  , KnownBy magicmirror dwarf 0 0 0 0 0 0
;
```

and see our new functionality in action:

```
>ask guard about snowwhite
There is not reply.

>tell guard about snowwhite
This provokes no reaction.

>ask dwarf about snowwhite
"Yes, I remember her. But that was long ago."
```

```
>tell guard about snowwhite
"Yes, I remember her. But that was long ago."

>ask guard about snowwhite
"Yes, I remember her. But that was long ago."
```

Separating Information and Conversation

A drawback to uniting information and knowledge is that we lose the ability to tailor a response to an NPC's personality. Separating the text that qualifies as topic information from the text that is character specific addresses this and creates dialog that is much more fluid and believable. Consider the ways in which each of the NPCs may differ in their answers regarding 'snowwhite,' as well as the differing ways that the topic could be broached:

```
You hesitate for a moment, then: "Do you remember Snow White?"
"Yes, I remember her," replies the dwarf with a dreamy
expression. "But that was long ago."  For a moment you think you
see a tear in his eye.
```

or

```
"Do you remember Snow White?" you ask.
The guard looks at you with disdain. "Yes, I remember her," he
says smugly. "But that was long ago."
```

or

```
You look around, making sure no one can see you talking to the
mirror. "Do you remember Snow White?"  you ask it, feeling
somewhat foolish. For a moment the mirror pauses, as though
trying to recall something almost forgotten.  Then it speaks:
"Yes, I remember her.  But that was long ago."
```

The narrative that accompanies dialog occurs in many different forms, but breaking the topic's information into, for example, two pieces can facilitate the use of most of these forms. The above examples can be accomplished by doing this. Text for asking about a topic also needs to be introduced:

```
KnowledgeTopic snowwhite magicmirror
  has learnable
  with name 'snowwhite' 'girl'
  , Query "Do you remember Snow White?" 0 !zero: don't replace punctuation
  , TopicInformation "Yes, I remember her" "." "But that was long ago" "."
  , KnownBy magicmirror dwarf 0 0 0 0 0 0
;
```

Notice that the ending punctuation has been separated from the rest of the text. In the English language (and shown in the above examples), periods are often substituted with commas when joining a piece of dialog with a piece of narrative. This separation gives us the option to replace the punctuation programmatically. It would be nice if the substitution always occurred, but with some punctuation, such as the question mark, the punctuation remains and is not replaced by a comma. To signify this, we can choose a zero value for the punctuation list element. Additionally, you can see that there are no quotation marks included in the information text. These will be included or not, by the narrative text.

Adding all of this new diversity to the information objects greatly increases the complexity. Since we may not always want the complexity we should be able to opt out of it if we so desire. A fairly good indicator that we don't want to personalize the text to the NPC is having only one value in the given property. That is, if the ending punctuation wasn't provided then just print the text and move on. The following KnowledgeTopic TellAbout routine, together with a new property, does a basic form of this conversation "blending" provided the telling NPC doesn't provide a means to do this:

```
... part of the KnowledgeTopic base class
  , TellAbout[to from;
     if((self.#topicinformation/2)>1){
        if(from provides ProcessDialog) {
           from.ProcessDialog(to, PersonalizeTell,
              self, TopicInformation);
        }
        else{
           if(from==player) print "You say, ";
           else print (The)from, " says, ";
           FlushDialog(self,TopicInformation,0);
        }
     }
     else PrintOrRun(self,TopicInformation);
     if(self has learnable) self.MemorizeFor(to);
  ]
```

Note that the ProcessDialog routine is called if the NPC provides it. This routine will parse the topic information and print it through a property we've specified, in

this case PersonalizeTell. We've made a presumption that if the object provides ProcessDialog, then it also provides PersonalizeTell. We could check for both, but if we add them to our NPC base class, then it is really not such a large leap of faith to presume that the existence of one implies the existence of the other.

```
... part of the NPC base class
  ProcessDialog[talkto procedure knobj info s1 p1 s2 p2;
     s1=(knobj.&info-->0);
     p1=knobj.&info-->1;
     s2=0; !default to nothing
     if((knobj.#info/2)>2){
        s2=knobj.&info-->2;
        p2=knobj.&info-->3;
     }
     self.procedure(talkto, s1,p1,s2,p2);
     FlushDialog(knobj,info,4); !print out remaining text
  ]
  , TellLine[before s1 canreplace p1 after;!print a single portion of text
        if(s1){print (string)before, (string)s1;
           if(canreplace)print (string)p1; !can replace punctuation?
           print (string)after;
           return true;
        }
        return false;
     ]
  , PersonalizeTell[talkto s1 p1 s2 p2; !override for npc formating
        if(self.TellLine("~",s1,p1,",","~ "));
        if(self==player) print "you say to ";
        else print (the)self," says to ";
        if(talkto==player) print "you. ";
        else print (the)talkto,".";
        self.TellLine("~",s2,p2,".","~ ");
     ]
```

The routine for wrapping all remaining text in quotes and printing it is best allocated globally since multiple classes will use it.

```
[FlushDialog knobj info start t ;
  if((knobj.#info/2)   print"~";
  for(t=start:t<(knobj.#info/2):t=t+2){
     if(t>start) print " ";
     print (string) knobj.&info-->t;
     if(knobj.&info-->(t+1)) print (string) knobj.&info-->(t+1);
  }
  print"~";
];
```

So why did we pass in PersonalizeTell rather than just calling it from the ProcessDialog? Doing so enables us to reuse the code in ProcessDialog and specify other routines for the text customization. We'll use this in a moment when our characters begin to "ask" as well as "tell."

Now we have the entire framework in place. Alone, we have far better than just quoted text.

```
>ask dwarf about snowwhite
"Yes, I remember her," the dwarf says to you. "But that was long
ago."
```

But we can also personalize the telling of topics with regard to the speaking NPC:

```
NPC -> dwarf "dwarf"
  with name 'dwarf'
  , PersonalizeTell[talkto s1 p1 s2 p2;
     talkto=0; !disable warning msg
     self.TellLine("~",s1,p1,",","~ replies the dwarf with
        a dreamy expression. ");
     self.TellLine("~",s2,p2,".","~ For a moment you think you see
        a tear in his eye.");
];

NPC -> guard "guard"
  with name 'guard'
  , PersonalizeTell[talkto s1 p1 s2 p2;
     print "The guard looks at ";
     if(talkto==player) print "you";
     else print (the)talkto;
     self.TellLine(" with disdain.  ~",s1,p1,",",
        "~ he says smugly. ");
     self.TellLine("~",s2,p2,".","~ ");
];

NPC -> magicmirror "magic mirror"
  with name 'magic' 'mirror'
  , PersonalizeTell[talkto s1 p1 s2 p2;
     talkto=0; !disable warning msg
     self.TellLine("For a moment the mirror pauses, as though
        trying to recall something almost forgotten.
        Then it speaks: ~",s1,p1," "," ");
     if(self.TellLine("",s2,p2,".","~ ")==false) print"~";
  ]
;
```

Of course, the PersonalizedTell routine works better if stocked with several variations of text, but this is a good starting point. To completely implement our examples, we still need to implement a PersonalizedAsk routine. Since the act of asking is done by the player rather than the NPC, it needs to be implemented somewhere other than the NPC's life routine. Replacing the standard library's AskSub routine is the obvious choice:

```
replace AskSub;

[AskSub;
  if(second ofclass KnowledgeTopic)
  second.AskAbout(noun, actor);
  if(RunLife(noun,##Ask)~=0) rfalse;
   L__M(##Ask,1,noun);
];
```

Now we can implement the AskAbout routine just as we did TellAbout:

```
...in the KnowledgeTopic class
  , AskAbout[askwho askby;
     if((self provides query)==false) return;
     if((self.#query/2)>1 && askwho has animate or talkable){
        if(askby provides ProcessDialog) {
           askby.ProcessDialog(askwho, PersonalizeAsk,
              self,query);
        }
     }
     else{
        if(askby==player) print "You ask, ";
        else print (The)askby, " asks, ";
        FlushDialog(self, query,0);
     }
  }
  else PrintOrRun(self,query);
  print "^";
]
```

And add the default PersonalizeAsk routine to the NPC base class:

```
... NPC class
  , PersonalizeAsk [talkto s1 p1 s2 p2; !over-ride for npc formatting
     if(self.TellLine("~",s1,p1,",","~ ")){
        if(self==player) print "you ask ";
        else print (the)self," asks ";
        if(talkto==player) print "you. ";
        else print (the)talkto,".";
     }
     self.TellLine("~",s2,p2,".","~ ");
  ]
```

At this point, we've accomplished everything we need to print out the query text when the player asks about a topic. We've also laid the framework for our NPCs to ask questions. Personalizing the narrative is accomplished for "ask" in the same way that it was for "tell." For example, let's add the following personalization to the guard:

```
...guard instance
  , PersonalizeAsk [talkto s1 p1 s2 p2;
     talkto=0; !disable warning
     if(self==player) print "You hesitate";
     else print (The)self," hesitates";
     self.TellLine(" for a moment, then:  ~",s1,p1,".",
        " ");
     if(~self.TellLine("~",s2,p2,".","~ ")) print"~";
  ]
```

The following code will now cause the guard to ask the dwarf about the snowwhite topic:

```
actor=guard;
<ask dwarf snowwhite>;
```

Upon execution, the generated text reads:

```
The guard hesitates for a moment, then:  "Do you remember
Snow White? "
"Yes, I remember her," replies the dwarf with a dreamy
expression. "But that was long ago." For a moment you think
you see a tear in his eye.
```

To top it all off, when all of this narrative diversity is said and done, the guard has now learned something that he can talk about if asked.

As you look over the guard's PersonalizeAsk routine, you might be wondering about the "You hesitate" text and the corresponding check to see if the guard and the player are one and the same. Of course, this is only necessary if you plan of letting the player "body jump" into the guard, and in a real game, we would want to make similar changes to the PersonalizeTell routine. As it happens, we can utilize the standard library's ChangePlayer() routine to demonstrate giving the player the same conversation abilities that we have given our NPCs:

```
<ask dwarf snowwhite>;
print "^     <Player becomes the guard...>^";
ChangePlayer(guard);
<ask dwarf snowwhite>;
```

Notice that when the player character becomes the NPC, he also inherits the personalization routines:

```
"Do you remember Snow White?"
"Yes, I remember her," replies the dwarf with a dreamy
expression. "But that was long ago." For a moment you think
you see a tear in his eye.
```

```
     <Player becomes the guard...>
You hesitate for a moment, then:  "Do you remember
Snow White?"
"Yes, I remember her," replies the dwarf with a dreamy
expression. "But that was long ago." For a moment you think
you see a tear in his eye.
```

A useful technique, even for games where the player does not change, is to create an NPC that will be the player. This way, questions asked by the player will have the same customized formatting as questions asked by NPCs.

It should be noted that the pronoun print rules discussed in the previous article " Pronouns On Steroids" really shine in this area and make coding dialog for these sorts of dual-purpose NPCs much simpler.

On to More Advanced Techniques

In this section we have covered several advanced topics, that can be used to customize text to a specific NPC. There's more to come in section 3...

Knowledge and Conversation
§3: More Advanced Techniques
(Advanced NPC Implementation, Part 2c)

n the previous section we discussed various methods of sharing knowledge between characters and customizing the dialog to match a character. This section will cover additional techniques, such as topics that change content.

Morphing Topics

A common practice when developing NPCs is to have the character say random things at various points in the game. Embedded somewhere deep in whatever daemon controls the NPC can be a couple of lines of code that cause the character to speak to the player:

```
actor=dwarf;
<tell player smalltalk>;
```

Although it is certainly possible to create a dozen or so KnowledgeTopics, pick one at random each turn and have the NPC talk about it, a better way is to create a morphing topic. That is, a topic that prints something different each time it is used. Since this technique is best implemented with non-learnable topics, separating conversation from information need not be done:

```
KnowledgeTopic smalltalk
  with name 'smalltalk'
  , KnownBy dwarf
  , TopicInformation[;
     switch(random(4)){
        1: "~There don't appear to be any other dwarves here,~
           says the dwarf.  ~I'll have to notify the
           labor board.~";
        2: "~I'm trying to keep my shoes clean,~ says the dwarf. ~My
           brothers are all envious of them.~";
        3: "~Hi-ho!~ spouts the dwarf.";
        4: "The dwarf scratches his bearded chin and says,
           ~Hmmm, I wonder where my children hid my hat.~ ";
     }
  ]
;
```

Additionally, using a variation of this technique we can customize conversations to follow a specific script...

Conversation Script Topics

One of the more difficult challenges that must be faced when writing believable NPC dialog via the ASK/Tell method is getting the player to ask the right questions. In many cases there is information that needs to be conveyed to the player to further the game, yet there is no viable method of forcing him/her to make the appropriate inquiries. It is true that the occasional hint can be dropped here and there, but there are often times that hints just don't seem appropriate to the game.

Having an NPC initiate conversation and thereby volunteer this information is a time-honored method of dealing with situation. In truth, this technique can be useful even if no pertinent information needs to be conveyed. People initiate conversation and volunteer trivial information as a matter of common practice. A realistic NPC should do this as well. Having an NPC follow a "conversation script" that evolves over a series of turns can help to simulate a conversation and provide pertinent information if needed.

There is a caveat or two associated with this approach, though. Many developers, myself included, have implemented this technique and rendered the player helpless for a time. That is, a situation has been created where the player can do nothing except wait until the NPC is finished talking. No amount of ingenious dialog can make up for a lack of interactivity. If the only action available to the player is to type "z", the game's "player experience" will suffer.

Another thing to consider when designing a conversation script is that people generally do not repeat the same information unless asked to. In a conversation where the player actually participates, it is possible that an NPC will be asked a question earlier than he was scripted to volunteer it. Repeating the information after it has been already been told is repetitive. What is needed is a way for the NPC to remember what has already been said and to move on to the next item in the script.

The "conversation script topic," a variation of morphing topics, is one of the more useful techniques in creating realistic, non-repeating dialog. It is particularly powerful because it can utilize other KnowledgeTopic objects so that the information can be both asked for and volunteered.

A couple of minor modifications should be implemented to facilitate knowledge scripts. The first is a method of recording whether or not a topic has been

communicated before. We can accomplish this by defining and setting an attribute. Arbitrarily, we'll call this 'hasbeentold':

```
attribute hasbeentold;
```

The next modification we can make will ensure that the NPC doesn't "double talk" or say two things at once. This can happen when the NPC is following a script and the player asks a question. If not coded correctly, the NPC will answer the question and then continue to say whatever comes next on the topic script. Game conversation seems to more closely parallel reality if the NPC only "drives" the conversation when the player isn't talking to him. To accomplish this, we can add a property to the NPC class that records the last move he/she spoke in:

```
...add to NPC class
  , LastMoveSpoken 0
```

These values can be updated when a topic is conveyed. The TellAbout routine, in the KnowledgeTopic class, is the obvious place:

```
...in the KnowledgeTopic class
  , TellAbout[to from;
     if(from provides LastMoveSpoken)
     from.LastMoveSpoken=turns+1;
     give self HasBeenTold;
     if((self.#topicinformation/2)>1){
        if(from provides ProcessDialog)
           from.ProcessDialog(to, PersonalizeTell,
              self, TopicInformation);
        else{
           if(from==player) print "You say, ";
           else print (The)from, " says, ";
           FlushDialog(self,TopicInformation,0);
        }
     }
     else PrintOrRun(self,TopicInformation);
     if(self has learnable)
     Self.MemorizeFor(to);
  ]
```

The final new addition that we will make to the KnowledgeTopic object, which we have been developing in this article, is a routine called SayOrRecurse. The sole purpose of this routine is to facilitate a conversation script. This routine, which is called by the TellAbout routine, checks to see if the specified KnowledgeTopic has

already been conveyed. If not, then it will print it and return. If so, then the TellAbout routine of the current object is called (recursively) to request the next scripted topic:

```
...from KnowledgeTopic class
  , SayOrRecurse [o to from;
     if(o hasnt hasbeentold) o.TellAbout(to, from);
     else self.TellAbout(to, from);
  ]
```

Opinions vary concerning the use of recursion. Suffice it to say, misuse of this technique can cause run-time errors in the Z-Machine interpreter. Although it can be a useful addition to your conversation repertoire, use this method with care.

Before creating a conversation script, an author should have a clear idea of how the conversation should flow. A design technique used by many authors is to write the entire conversation the way they would prefer it to read first. Once this has been done, the conversation can be logically separated into pieces. Almost always, these pieces of text can be classified into two categories: dialog that is the answer to a question which the player will possibly ask, and dialog that will never be asked about, and serves no purpose other than to make the NPC seem more believable.

The former type, which can be inquired about, is best stored in a KnowledgeTopic of it's own. Generally, these types of dialog pieces will compromise the bulk of a conversation script. The latter type more often appears at the beginning and end of a script. Generic statements, such as variations of "hello," and "goodbye", are classified in this category.

What follows is an example script object for a conversation between a doctor and the player who has just woken up in a hospital. Notice that script steps 2, 3, and 4 utilize three other KnowledgeTopics and so will be brought up, even if not directly asked about by the player.

```
KnowledgeTopic DocScript_t
  with     count     0, KnownBy doctor
  , TellAbout[towho fromwho;
     self.count=self.count+1;
     switch(self.count) {
        1: print "^~How are you feeling, Mr. Valentine?  I'm pleased to
              see you've awakened,~ the man says to you. ";
        2: self.SayOrRecurse(doc_who_t,towho, fromwho);
```

```
        3: self.SayOrRecurse(doc_me_t, towho, fromwho);
        4: self.SayOrRecurse(hosp_where_t, towho,fromwho);
        5: print "~Okay then,~ says the doctor as he opens the door.  ~If you
              have anymore questions, feel free to buzz the nurse.~  He
              leaves and closes the door behind him.";
           self.TellAbout=NULL;
     }
  ]
;
KnowledgeTopic doc_who_t
  with     name 'him' 'himself' 'man' 'doc' 'doctor' 'who'
  , KnownBy doctor
  , query  "^~So who are you?~ I asked the man. "
  , TopicInformation "^~My name is Doctor Harris.  I'm the resident
     Nuerological specialist.~ ^^~Nuerological.~ You repeat,
     then touch your head probingly. No bandage.^^The doctor
     nods. "
;
KnowledgeTopic doc_me_t
  with name 'myself' 'self' 'me' 'head' 'damage' 'condition'
  , KnownBy doctor
  , query "^~So what's my condition, Doc?~ You ask."
  , TopicInformation "^^~You're the talk of the hospital right now,~ he replies.
     ~You've been comatose for several weeks. CAT scans showed
     extreme cerebral dysfunction. To be frank, the prognosis was
     bleak.^^~This morning, for no apparent reason, monitors
     registered a return to normal brain activity. Further examination
     showed burned tissue and broken ribs healed. I've never seen
     anything like it before.~^^Cerebral dysfunction... You think
     about that for a moment when it suddenly dawns on you that
     you cannot remember anything about yourself, including your
     own name!"
;
KnowledgeTopic hosp_where_t
  with name 'where' 'hospital' 'location'
  , KnownBy doctor
  , query "^~So where am I?~ ,You ask the doctor. "
  , TopicInformation "^~You're at Saint Augustine Hospital.~^Saint Augustine.
     You don't recognize the name. "
;
```

Additionally, a believable NPC will have a collection of additional topics that are not volunteered, but are clarifications of statements made in the script. This NPC will be able to answer questions the player may have in the course of the conversation but will not volunteer this information unless directly asked about it:

```
KnowledgeTopic doc_amnesia_t
  with name 'memory' 'amnesia'
  , KnownBy doctor
  , query "^~So could my memory be affected?~ You ask."
  , TopicInformation "^^~Certainly,~ he said.
     ~Without a doubt, the explosion, and
     events immediately preceding and following it would be
      unclear to you. That's normal.  There may even be
      more substantial memory loss.~"
;
KnowledgeTopic explosion_t
  with name 'explosion'
  , KnownBy doctor
  , query "^~What happened? You said I was in an explosion? ~"
  , TopicInformation "^^A sad look comes across the doctor's face.
     ~I'm afraid that I don't know many of the details behind what
     happened to you. I only know that there was an explosion of some kind.
     I have no information as to the condition of your family.~^^You watch
     his eyes as he says this to you. For reasons you don't know, you
     concentrate upon pupils as they dilate. In the back of your head, some
     forgotten memory tells you that this means he is lying. "
;
```

Now that we have a workable script, we need an NPC capable of driving the conversation:

```
NPC -> doctor "man"
  with name 'doc' 'doctor' 'harris' 'man'
  , description "The man is dressed in a white coat.  He
     appears to be a doctor."
  , each_turn[; if(self.LastMoveSpoken==turns) return;
     if(DocScript_t provides TellAbout)
     DocScript_t.TellAbout(player,self);]
;
```

This simple NPC will converse with the player each turn about our script object. The conversation can happen in several different ways now. One possible play out of the script follows:

```
"How are you feeling, Mr. Valentine? I'm pleased to see you've
awakened," the man says to you.

>x man
The man is dressed in a white coat. He appears to be a doctor.
"My name is Doctor Harris. I'm the resident Nuerological specialist."

"Nuerological." You repeat, and then touch your head probingly. No bandage.
The doctor nods.

>ask doc about head
"So what's my condition, Doc?" You ask.
"You're the talk of the hospital right now," he
replies. "You've been comatose for several weeks. CAT
scans showed extreme cerebral dysfunction. To be frank, the prognosis
was bleak.

"This morning, for no apparent reason, monitors registered a return
to normal brain activity. Further examination showed burned tissue and
broken ribs healed. I've never seen anything like it before."

Cerebral dysfunction... You think about that for a moment when it suddenly
dawns on you that you cannot remember anything about yourself, including
your own name!

>ask doc about hospital
"So where am I?", You ask the doctor.
"You're at Saint Augustine Hospital."

Saint Augustine. You don't recognize the name.

>ask doc about memory
"So could my memory be affected?" You ask.
"Certainly," he said. "Without a doubt, the explosion, and events
immediately preceding and following it would be unclear to you. That's
normal. There may even be more substantial memory loss."

>ask doc about explosion
"What happened? You said I was in an explosion?"
A sad look comes across the doctor's face. "I'm afraid that I
don't know many of the details behind what happened to you. I only know
that there was an explosion of some kind. I have no information as to the
condition of your family."

You watch his eyes as he says this to you. For reasons you don't know,
you concentrate upon his pupils as they dilate. In the back of your head,
some forgotten memory tells you this means he is lying.
```

```
>z
Time passes.
"Okay then," says the doctor as he opens the door. "If you have
anymore questions, feel free to buzz the nurse."  He leaves and closes
the door behind him.
```

Of course this is just a basic example, but notice that the script adapts and skips over topics already asked about by the player. This technique has the advantage of ensuring that the player gets information, even if he does not think to ask the appropriate question, but also adapts to a player's questions, ensuring a more realistic conversation.

Inanimate Knowledge Transfer

It was mentioned above that characters are not the only objects that can utilize the knowledge objects. Any book, such as "Waldecks's Mayan dictionary" from the Inform Designer's Manual, or the ever-popular spell book is an ideal candidate for an inanimate object housing KnowledgeTopics. The consult verb should be extended much as the tell verb was:

```
[ConsultTopicSub; second.TellAbout(actor,noun); ];

extend "consult" first * noun 'about' scope=TopicInTarget-> ConsultTopic
  * noun 'on' scope=TopicInTarget -> ConsultTopic ;
```

Now we can create the spell book with a specialized ProcessDialog routine, specifically tailored to a reference book:

```
Object -> spellbook "dusty old spell book"
  with name 'spell' 'book' 'dusty' 'old'
  , ProcessDialog [talkto procedure knobj info;
     talkto=procedure; !warning supression
     print "Flipping through the old book uncovers the following entry:^";
     FlushDialog(knobj,info,0);
  ]
;
```

And of course a spell for the spell book:

```
KnowledgeTopic lightspell
  has learnable
```

```
   with name 'lumos' 'light'
   , KnownBy spellbook 0 0 0 0 0 0 0 0 0
   , TopicInformation "The lumos spell provides light" "."
;
```

Since the spell book is not a "creature," as defined by the standard library, we cannot tell the spell book anything (this is a good thing), and so the spell book cannot learn new KnowledgeTopics as NPCs do.

In a sense, a journal can "learn" about a topic when someone writes in it. A good exercise for the reader would be to implement a "write" verb in which knowledge can be recorded in non-animate objects.

In Closing

The reader may have noticed that, although Ask and Tell are thoroughly discussed in this article, Answer (or say) has been left relatively untouched. It has been my experience that it is often best to treat Answer and Tell as two forms of the same command. Essentially, I choose to treat:

```
>Say snowwhite to dwarf
```

in the same manner as I would treat:

```
>tell dwarf about snowwhite
```

As stated previously, my way is not the only way. Distinguishing between the two commands may be useful in a number of scenarios. The above "spell book" example, set in a world where characters cast spells, would be one such situation. For example,

```
>say lumos
```

would be better handled in the same manner as

```
>cast lumos
A bright light appears and brightens the room.
```

instead of

```
>tell dwarf about lumos
You give the dwarf a condescending gaze.  "The lumos
spell provides light,"  You say.
```

This distinction is best made by the developer.

Also, in the interest of avoiding dependencies, all references to specialized print rules have been left out of the samples above. Pronoun print rules, however, are especially useful when creating the sort of conversation topics that we have been developing in this article. Techniques discussed in the previous article "Pronouns on Steroids" can be of significant aid.

About the Examples...

To aid the reader, the examples given in all three sections of this article have been bundled together in single source file.

Additionally, the ORLibrary entries ORKnowledgeTopic.h and ORNPC.h contain a complete, and somewhat more progressive, implementation of the techniques discussed in this article.

For review

Emily Short's page on NPC design, although geared for platform independence and not for Inform specifically, is nevertheless inspirational in its discussion.

Additionally, her game "Galatea", is an exquisite example of NPC conversation.

Also, although mentioned previously, the conversation section of Roger Firth's Infact page is especially recommended.

Advanced NPC Implementation, Part 3
NPC Actions

erb subroutines do it all. They enforce gravity; they differentiate between light and darkness; they decide where a character can travel. These little routines implement the laws of physics in our game world and govern what can and cannot be done. Yet for some reason, historically they only affect the player character. The game world laws that restrict the player do not bind NPCs. The following code exemplifies:

```
object room "room"
  has light
  with description "This is an empty room."
;
object->"guard" has animate
  with name 'guard'
  , orders [;
     take: move noun to self;
        print_ret (The)self," takes ",(the)noun,".";
  ]
;
object->"table" has static supporter with name 'table';
object->->"apple" has edible with name 'apple';
```

The guard has now been coded to respond to the "take" order and does so too well. In the above example, he will successfully "take" pretty much anything in scope, even things that he should not be able to:

```
room
 This is an empty room.

 You can see a guard and a table here.

 >take table
 That's fixed in place.

 >guard, take table
 The guard takes the table.

 >guard, take self
 The guard takes yourself.

 It is now pitch dark in here!
```

Obviously this sort of "empowered taking" holds a very narrow slice of the "desired NPC abilities" pie. By convention, the filtering of what an NPC cannot do is also coded in the appropriate routine:

```
... replace the guard's previous orders routine
  , orders [;
     take:
        if(noun hasnt static && noun~=player){
           move noun to self;
           print_ret (The)self," takes ",(the)noun,".";
        }
        print_ret (The)self," cannot take that.";
     ]
```

For the majority of Inform works, that's the way NPC action is accomplished. Piles and piles of example source code and sample NPCs follow this convention of manipulating the game world directly. There are some drawbacks to this, however:

- This technique is repetitious. Putting the same rules in the NPC's reaction routines duplicates code that exists in the library's verb routines.
- This technique can be overly complex. The library considers numerous variables in its efforts to implement the game world. It is very easy to overlook details when writing this sort of code. For instance, in the above example, the guard cannot pickup the player or objects with the "static" attribute, but there is nothing stopping him from taking either "untouchable" objects, or other animate objects. For instance, the guard could take a key contained inside of a locked, glass box (transparent) without breaking the glass. He could also take another NPC behind a locked, cell door. To imitate the library, we would have to code for these circumstances as well.
- New verbs, such as "cast spell" or "mount horse" are unavailable to the NPC without additional coding too.
- This technique does not respect an object's "before" routines, which may interrupt an action.
- By its very nature, this technique limits NPC design, since everything that an NPC can do must be coded for explicitly.

Fear not, for there is another, more powerful way...

Redirecting to Verb Routines

A more intuitive solution is to leverage the verb routines themselves. Successful implementation of this method addresses all of the above drawbacks and actually makes the code more readable. At first glance, it seems a deceptively simple means of implementing NPC actions. Many new developers have posted questions to the RAIF newsgroup asking what was wrong with their code and why redirecting actions doesn't seem to work. The following is an example or redirecting an ordered action using the library's "action and return" operator (<<...>>):

```
object->"guard" has animate
  with name 'guard'
  , orders [; take: <<take noun>>;]
;
```

In the initial test of this example, everything appears to work fine:

```
room
 This is an empty room.

 You can see a guard and a table (on which is an apple) here.

 >guard, get apple
 Taken.
```

It is not until you look with a little more scrutiny that the flaw in this approach appears:

```
 >i
 You are carrying:
   an apple
```

The library's verb routines presume that the player carries out all actions. The player is therefore the one that actually picked up the apple. This is unfortunate, but not insurmountable.

Actor, Good; Player, Bad

The standard library is glorious. It is, perhaps, the most useful collection of IF routines ever created. To look upon the Library's code is to be humbled.

Now then, having appropriately praised the library, we can now proceed with a slightly more critical discussion of it. In particular, if there is one area where I feel the library should have been written differently, it is library verbs and messages. Specifically, these two aspects of the library rely upon the "player" variable, rather than the "actor" variable. See how forcing the player variable to momentarily equal the actor variable almost fixes the above behavior:

```
... replace the guard's previous orders routine
  , orders [save retval;
     save=player;
     player=actor;
     <take noun>;
     player=save;
     return retval;
  ]
```

```
>guard, get apple
Taken.

>i
You are carrying nothing.
```

Of course this is simply a patch, and it prevents us from completing our implementation of NPC actions, as we'll see below. A legitimate solution would replace the library's verb and verb-helper routines with slightly modified versions that check the actor variable rather than the player variable. We can do this pretty easily to the "take" and "eat" verbs by cutting-and-pasting the appropriate routines from verblibm.h and then running a search-and-replace against them to change references to "player" into references to "actor":

```
!only slightly changed from the standard library
[ TakeSub;
  if (onotheld_mode==0 || noun notin actor)
  if (AttemptToTakeObject(noun)) rtrue;
  if (AfterRoutines()==1) rtrue;
  notheld_mode=onotheld_mode;
  if (notheld_mode==1 || keep_silent==1) rtrue;
  L__M(##Take,1);
];
[ EatSub;
  if (ObjectIsUntouchable(noun)) return;
  if (noun hasnt edible) return L__M(##Eat,1,noun);
  if (noun has worn){
     L__M(##Drop,3,noun);
     <Disrobe noun>;
     if (noun has worn && noun in actor) rtrue;
  }
  remove noun;
  if (AfterRoutines()==1) rtrue;
  if (keep_silent==1) rtrue;
  L__M(##Eat,2,noun);
];
[ AttemptToTakeObject item   ancestor after_recipient i j k;
  ...also needs to be updated since it is called from TakeSub
];
[ ObjectIsUntouchable item flag1 flag2 ancestor i;
  ...also needs to be updated since it is called from EatSub
];
```

As shown above, both TakeSub and EatSub call additional routines that also need to be updated in the same fashion. These are AttemptToTakeObject and ObjectIsUntouchable. We won't reprint the modified versions of these two routines since they are long and the changes are minute, but the point should be made that a simple search-and-replace fixes these as well.

Adding the four modified routines to your source code and putting the appropriate replace directives at the top makes the take and eat verbs available to the NPC without having to change the value of the player variable.

```
replace  TakeSub;
replace  EatSub;
replace  AttemptToTakeObject;
replace  ObjectIsUntouchable;
```

Also, ObjectIsUntouchable is called by numerous other verbs, and TakeSub is called by derivatives verbs. By replacing these four routines, we have also automatically made available to NPCs, all of the following routines and thus their corresponding verb forms:

```
TakeSub
EatSub
PullSub
PushSub
TurnSub
LockSub
UnlockSub
SqueezeSub
SwitchOnSub
SwitchOffSub
EmptySub
RemoveSub
DisrobeSub
WakeOtherSub
```

Since so many verbs are available to the NPC we will now drop the use of the library's "action" operators (and >) in favor of the library's more generic routine, ActionPrimitive. This allows us to forego addressing the implied switch(action) statement that exists in all "reaction" routines (such as orders, life, before, etc...) and redirect all orders to the appropriate verbs with a single line of code:

```
... replace the guard's previous orders routine
  ,   orders [; return ActionPrimitive();]
```

As we now have the ability to have the guard act for us, we can proceed with ordering him about and testing the affect of using "actor" instead of "player" in verb routines. Notice how drop, which has not yet been updated, does not function

correctly:

```
>guard, get apple
Taken.

>guard, drop apple
You haven't got that.

>guard, eat apple
You eat the apple. Not bad.
```

What's with "YOU eat the apple?" It's understandable that the guard could not drop the apple since DropSub has not been updated, but why didn't the guard eat the apple?

Morphing Library Messages

Actually, the guard did eat the apple, but the default library message for the verb "eat" was written expecting the PC to be the only character eating. It, and many more library messages, will need to be changed to accommodate this whole new world of NPC actions. Usually, the recommend way of overloading a library message is to declare a LibraryMessages object between "Parser.h" and "Verblib.h" and implement it in the before routine. The following replacement for the "eat" message demonstrates this approach:

```
object LibraryMessages
  with before[;
       Eat:
          if(lm_n==2){
             if(actor==player) print "You eat ";
             else print (The)actor," eats ";
             print (the) noun, ".";
             if(actor==player) print" Not bad.";
             return;
          }
       ];
```

Notice that the message now changes form depending on who is performing the action:

```
>guard, take apple then eat it
Taken.
The guard eats the apple.
```

or,

```
>take the apple then eat it
Taken.
You eat the apple. Not bad.
```

The number of library messages that need to be changed in order to implement NPC actions is somewhat excessive. Because of this, duplicating the language definition file (English.h) and making modifications there (specifying the modified file on the Inform command line or in the ICL file) may be a preferable method to instantiating a LibraryMessages object.

Initiating NPC Actions

For the examples given above, we have been calling ActionPrimitive() from within the NPC's orders routine. This has been convenient because all of the necessary library variables have already been set up for us. When the player is not issuing orders at run-time, however, we need to set these values ourselves. Additionally, there are another couple of complications that we must address:

```
1) The ActionPrimitive() routine does not honor the "before" property routines like
"action notation" does (<<action>>); In order to achieve this we mush make the call
ourselves.


2) Floating objects may not be in scope for the actor.
```

These issues can be addressed in a common routine, which seems to be most at home as part of an NPC base class:

```
class NPC
  with DoVerb[act n s svAL svActr svActn svN svS svInp1 svInp2 retval;
        svAL=actors_location;
        svActr=actor;
        svActn=action;
        svN=noun;
        svS=second;
        svInp1 = inp1;
        svInp2 = inp2;
        actors_location=parent(self);
        actor=self;
        action=act;
        noun=n;
        second=s;
        inp1 = noun;
        inp2 = second;
        MoveFloatingObjects();
        retval=BeforeRoutines();
        if(retval==false) retval=ActionPrimitive();
```

```
        actors_location=svAL;
        actor=svActr;
        action=svActn;
        noun=svN;
        second=svS;
        inp1 = svInp1;
        inp2 = svInp2;
        MoveFloatingObjects();
        return retval;
];
```

Additionally, BeforeRoutines() and AfterRoutines(), and MoveFloatingObjects() should all be modified in the same fashion as was discussed above. One exception to this "player equals actor" rule is in the call to the player's ORDERS property in the BeforeRoutines() method. This section of code should only be run if the player and actor are the same, otherwise an endless recursion results the will blow the interpreter's stack. A slightly modified BeforeRoutine() which addresses this follows:

```
[BeforeRoutines;
   if(GamePreRoutine()~=0) rtrue;
   if(actor==player && RunRoutines(player,orders)~=0) rtrue;
   if(location~=0 && RunRoutines(parent(actor),before)~=0)
         rtrue;
   scope_reason=REACT_BEFORE_REASON; parser_one=0;
   SearchScope(ScopeCeiling(actor),actor,0);
   scope_reason=PARSING_REASON;
   if(parser_one~=0) rtrue;
   if(inp1>1 && RunRoutines(inp1,before)~=0) rtrue;
   rfalse;
];
```

Assuming our NPC is derived from the NPC class, this addition will allow us to direct NPC actions from any section of code (like a controlling daemon):

```
 !equates to "guard, put apple on table"
 guard.DoVerb(##PutOn, apple, table);
```

A Peek Inside Pandora's Box

Perhaps the reason this technique is not often seen is the number of convoluted changes that need to be made in order to implement it. At times, implementation of this technique can seem horribly daunting.

The examples given above are far from complete. In addition to the excessive number of messages that must be modified, a quick scan through the verblibm.h

file uncovers the following verb routines that need to be updated (only two of these were done in the above examples):

```
InvSub
TakeSub
DropSub
PutOnSub
InsertSub
TransferSub
GiveSub
ShowSub
EnterSub
GetOffSub
ExitSub
GoSub
LookSub
ExamineSub
LookUnderSub
ExitSub
TouchSub
WaveSub
PushSub
KissSub
ThrowAtSub
TellSub
AskSub
OpenSub
WearSub
EatSub
```

There are also several additional routines what are called by these verb subs that also need to be updated. Further, in many of these routines, there are references to the library variables "location" and "real_location." These need to be changed to the library variable "actors_location."

Obviously a complete implementation of this technique requires major modifications to the library, but once done allows for greater flexibility with simplified, yet more powerful, coding capabilities. Additionally, since the "actor" variable and the "player" variable are normally the same, these changes are transparent. That is, the new behavior only occurs when the new functionality is used and does not affect existing code. Without calling NPC actions, games run as they always have.

A Note on Code

The examples above were given to demonstrate this technique, but they do not necessarily represent the best way to implement it. The definition of what is "best" is left up to the reader. As a suggestion, however, rewriting messages to change form can be a convoluted and monotonous task. The print rules discussed in §1 of this article series "Pronouns on Steroids" are particularly useful.

As a final point, it should be noted that a complete implementation of the techniques discussed in this article can be found at the ORLibrary in the entries ORNPCVerb.h, OREnglish.h, and ORNPC_doverb.h.

A special thanks goes out to Stephen Robert Norris, whose input and bug fixing in the ORLibrary necessarily impacted this article.

Part F
Tutorials

Coming soon... A tutorial based on the Wade Wars!

Part G
Informed Tips

Randomizing Random

rom time to time, while reading RAIF postings, the subject of repeating random numbers comes up. For all appearances, this seems to be some sort of bug in one of the most popular and widely used Z-Code interpreters available today: Frotz. Reports seem to indicate that it affects various versions of Frotz. While researching this issue and developing a work around, I was able to confirm the behavior with WinFrotz version 2.32 release 5.3.

As an IF player, one might be tempted to simply change to a different interpreter, but a developer doesn't have that option. One of the beauties of the Z-Code standard is that it allows the players to choose their own interpreter. And to be perfectly frank: Frotz is too darn popular to ignore.

Show me the bug!

Before delving into the explanation of why this behavior occurs and how to fix it, let me include a code snip to demonstrate the bug (note: the standard library isn't really needed here to demonstrate this bug, but I prefer that sample code resemble real-world scenarios as closely as possible.)

```
Include "Parser"; Include "VerbLib"; !library includes
Object    SimpleRoom "Simple Room" !single room
 with   description "This is a simple room"
 has    light
;
[Initialise;
 random(0);   !Z-code standard says
          !that this command is
          !SUPPOSED to initialize
          !the random number generator
 location=SimpleRoom;
];
Include "Grammar"; !library include
Verb "rand" !!! an example rand command
 * -> rand
;
[randsub t; !implementation of new verb
 for(t=1:t<10:t++)
 print random(100),"^";
];
end;
```

This is a pretty straightforward example. One room and a grammar rule the prints ten random numbers. Compile this, and run the resulting .Z5 file under Frotz. At

the command prompt enter in the command "RAND" and you'll see the ten random numbers printed. Looks good so far.

Now write down these ten numbers and enter the command "RESTART". At the new command prompt enter "RAND" again. If your interpreter suffers from this bug, then the new list of ten numbers will match the first list exactly.

Now this method of checking for the behavior will always demonstrate it if it exists. Period. But the behavior will often pop up in less predictable ways. For instance, it is possible to:

```
1) start Frotz
2) run this example
3) record the ten "random numbers"
4) exit Frotz entirely
5) start Frotz again
6) run the example again...
```

...And still get the same random sequence. Because of the way random numbers work, this method will not always be the same. But often they will.

But why? Aren't random numbers random?

No, actually they are not. All random numbers generated by modern computers are actually pseudo-random numbers, which are the result of a mathematical formula that is applied against the previously produced number.

If you didn't catch that don't worry about it. Think of it like this: That list of ten numbers that was generated in the example program above is associated with another number called a "seed" value. Behind the scenes, the interpreter has the job of setting this seed value. This called "seeding the random number generator." Seeding the generator with the same value will always produce the same list. That doesn't make for a very random program. Traditionally, to achieve a non-repeating sequence every run, the random number generator is seeded with the current time represented in seconds elapsed since 1980.

In Inform, this is not possible since 1) there does not appear to be a way to seed the generator even if we had a seed value, and 2) there is not a readily available method that will return a non-predictable result to use as a seed value. According to the Z-code standard, calling random(0); is supposed to accomplish both of these, but as is the case with WinFrotz, not all interpreters have this implemented.

The Fix...

Luckily, there is a way of accomplishing the first task's desired effect without "seeding" the generator at all. Specifically, cycle through the random-number generator a random number of times. Given a single static list of randomly generated numbers, a non-predicatable sequence can be achieved simply by starting at a random location in the list. Think of this as pseudo-seeding the pseudo-random number generator.

"What's that?" you say. "Random numbers are what we are unable to get here. How do you get a random number to use for iterating through the generator with?" You don't. Instead you choose a number that will almost always be different every time you run. Some programmers opt to use a value equal to the current number of game moves elapsed once the player performs some arbitrary task (like entering a room). But that still leaves the game predictable, just less so.

The solution shown below measures the only thing that is bound to be different every game: the length of time it takes a user to enter text and press return in milliseconds!

This solution also has the advantages of being both reusable and transparent. It is reusable in that it is not game specific. It isn't tied to a specific event or action. It works with any game without modification. It is transparent in that the developer has to do nothing other than include the text. No special reference needs to be made to it. No methods need to be called. The block of code can be implemented at the top of any game source file and work. It's just drop and go code.

```
REPLACE KeyboardPrimitive;
global randseed=0;
[KeyboardPrimitive a_buffer a_table t;
 if(randseed>0) !Already randomized,
             !so use library code
    read a_buffer a_table;
 else !otherwise, get and randomize
 {
    @aread a_buffer a_table 1 procrandseed t;
    for(t=0:t<randseed:t++) random(1);
 }
];
[procrandseed;
 randseed++;
 return false;
];
```

Just cut and paste this code to the top of the example code we worked with earlier. Recompile and run the tests again to see the behavior corrected.

Making VERBOSE the default mode

The Inform Library provides three commands for players to control how rooms are described: VERBOSE prints a room's full description every time you go there, BRIEF prints the full description only on your first visit, and SUPERBRIEF never prints the full description.The Library's default setting is BRIEF, but many authors (and players) prefer VERBOSE behaviour -- to have a room fully described on each visit. Of course, just typing a VERBOSE command achieves this, but it's annoying to have to remember every time: much more convenient to make VERBOSE the standard setting.The Library uses a global variable **lookmode** which can take the values 1 (BRIEF), 2 (VERBOSE) and 3 (SUPERBRIEF), and w ich is initialised to 1 in the `parserm.h` file. So, to make VERBOSE the default, you can either just edit that file, or better (since it's preferable where possible to over-ride the Library rather than change it) add this line to your **Initialise()** function:

```
lookmode = 2; !! default is VERBOSE mode
```

At a mimimum, that's all you need to do. However, here are three additional changes that you may wish to incorporate. First, the message printed by the BRIEF command starts off:

```
YOURSTORY is now in its normal "brief" printing mode...
```

when of course BRIEF now *isn't* the normal mode. Second, the command NORMAL should be a synomym for VERBOSE rather than for BRIEF. Finally, you may consider that players should always see a room's full description at least once; therefore, you wish to disable entirely the SUPERBRIEF command. Here's the skeleton of a game which does all this:

```
Constant Story "MyStory";
Constant Headline "^An example game.^";

Include "Parser";

Object  LibraryMessages with before [; !! replace three Library messages
  LMode1: " is now in its ~brief~ printing mode, which gives long descriptions
            of places never before visited and short descriptions otherwise.";
  LMode2: " is now in its normal ~verbose~ mode, which always gives long
            descriptions of locations (even if you've been there before).";
  LMode3: " doesn't support ~superbrief~ mode.";
        ];
```

```
Replace LMode3Sub; !! don't use original Library routine

Include "VerbLib";

[ Initialise;
        lookmode = 2; !! default is VERBOSE mode
        location = hall;
        "^^Adventures don't get much simpler than this.^^";
        ];

[ LMode3Sub; print (string) Story; L__M(##LMode3); ]; !! replacement routine

Class   Room
  has   light;

Room    hall "Hall"
  with  description "A door leads south from the hall...",
        s_to study;

Room    study "Study"
  with  description "A door leads north from the study...",
        n_to hall;

Include "Grammar";

Extend only 'normal' replace * -> LMode2; !! NORMAL = VERBOSE
```

Using TextPad with Inform

TextPad is a splendid text editor for Windows: powerful, packed full of useful features, and still easy to use. It's available as a shareware download from www.textpad.com. While there, also visit www.textpad.com/add-ons/to download `txplink4.zip` (which fixes Windows so that attempts to call the much inferior NotePad simply fire up TextPad instead), and www.textpad.com/add-ons/hmsyn.htmlto fetch `inform2.zip`.

Once you've downloaded those files, do this:

- run the downloaded `tpeng`*`NNN`*`.exe` and follow its install process,
- unzip `inform2.zip`, placing the resulting `inform2.syn` in the `TextPad\Samples` folder, and
- unzip `txplink4.zip` and follow the setup instructions therein.

Even if you do nothing else, you've got yourself a crackin' good editor. However, it's very straightforward to teach TextPad some tricks which will greatly enhance your Inform productivity.**Creating a TextPad document class**

A document class is a family of files having the same extension: for Inform, that's `.INF`. Follow these steps:

- Start TextPad.
- From the menu bar, select **Configure > New document class...** to display the Document Class Wizard.
- Enter a class name of **Inform**, and then click **Next**.
- Enter a class membership of ***.INF**, and then click **Next**.
- Click to enable syntax highlighting, select `Inform2.syn` from the dropdown list, and then click **Next**.
- Click **Finish**.
- Select **Configure > Preferences...**, click to open the list of **Document Classes**, and click to open **Inform**. Specify how you want your game files to appear.
- Click **OK**.

Now, any Inform source file that you edit will automatically use syntax highlighting and the other display preferences that you've set. Easy!**Running the compiler within TextPad**

Here's how to configure TextPad so that a single keystroke automatically compiles the file that you're editing. Follow these steps:

- Start TextPad.

- From the menu bar, select **Configure > Preferences...**, and click **Tools**.
- From the dropdown **Add** list, select **Program**.
- In the Select File dialog, navigate to the folder containing `infrmw32.exe`, select it, and then click **Open**.
- In the Preferences dialog, click **Apply**.
- Click to open the list of **Tools**, and click **Infrmw32**.
- Change the Parameters to (for example) **+include_path=.\,.\Library,..\Library $File**. This tells the compiler to look for Included files in the current folder, then in a Library sub-folder, and finally in a Library folder alongside the current folder. Change these settings to suit your personal preference.
- Glulx only: You may also need one other change to get error message handling to work properly; see the note below.
- Click **OK**.

If you select **Tools** from the menu bar, you'll see **Infrmw32**has appeared as the last item, with a keyboard shortcut of Ctrl+1. Try it: open an Inform file in TextPad, hold down the Ctrl key, and press 1. If you're lucky, you'll see "Tool completed successfully". More likely, you'll get some compiler error messages. Double-click on one of those messages to jump straight to the appropriate line in your Inform file.

Running the interpreter within TextPad

Use exactly the same process so that a single keystroke runs the interpreter. Follow these steps:

- Start TextPad.
- From the menu bar, select **Configure > Preferences...**, and click **Tools**.
- From the dropdown **Add** list, select **Program**.
- In the Select File dialog, navigate to the folder containing `WinFrotz.exe`, select it, and then click **Open**.
- In the Preferences dialog, click **Apply**.
- Click to open the list of **Tools**, and click **WinFrotz**.
- Change the Parameters to **$BaseName.Z5**.
- Click **OK**.

Now, if you select **Tools** from the menu bar, you'll see **WinFrotz**with a keyboard shortcut of Ctrl+2. Try it: open a valid Inform file in TextPad, compile it (Ctrl+1) close the "Success" window and then press Ctrl+2 to open it in the interpreter.

The instructions here set up TextPad to compile and run a Version 5 game. Using exactly the same processes, you could also configure Ctrl+2,3 to handle Version 8 games, and Ctrl+5,6 for Glulx. These are left as exercises for the reader.If you wish to change the Ctrl+1 and Ctrl+2 keyboard shortcuts, follow these steps:

- Start TextPad.
- From the menu bar, select **Configure > Preferences...**, and click **Keyboard**.
- From the Categories list, click **Tools**.
- Select the command and the current keys, use **Remove** and **Assign** to set a new shortcut.
- Click **OK**.

Note on Glulx error messages
(I'm indebted to Petar Kanuritch for this information.) The Inform and Glulx compilers can output their compilation results in three different styles:

- **-E0**: Archimedes-style error messages
  (for example: `line 427: Error: No such constant as "top_object"`)
- **-E1**: Microsoft-style error messages
  (for example: `C:\Glulx\libg610\Balances.inf(427): Error: No such constant as "top_object"`)
- **-E2**: Macintosh MPW-style error messages
  (for example: `File "C:\Glulx\libg610\Balances.inf"; Line 427 # Error: No such constant as "top_object"`)

where **-E***n* is the switch which sets the related style of error message. For each one, TextPad requires a different Regular Expression to match the output and enable a jump to the correct line. These are:

- **-E0 setting**
  ^line \([0-9]+\):
  File=
  Line=1
  Column=
- **-E1 setting**
  ^\([^(]+\)(\([0-9]+\)):
  File=1
  Line=2
  Column=

- **-E2 setting**
  ^File "\([^(]+\)"; Line \([0-9]+\)
  File=1
  Line=2
  Column=

The default entered by TextPad in the "Regular Expression to match output:" box is, happily, exactly the **-E1** setting; even better, Inform defaults to **-E1** style on a PC, so it all works nicely without further ado. However, Glulx defaults to **-E0**, so either (a) change TextPad's regular expression to the **-E0** setting, or (b) include **-E1** in TextPad's setting for Parameters.

REPLAYing command scripts

A game's development process involves a great deal of highly-focused testing. As you create each new room, object, NPC, verb and so on, you give it lots and lots of exercise, both to confirm that it does what it's supposed to, and to ensure that it doesn't fail in unusual circumstances. Doing this carefully as you go along is common sense: it's much easier to thrash the hell out of some new code while you can still clearly remember just how you intended it to behave.As the game grows, it becomes harder and harder to check it thoroughly. Sure, it's still easy to test the new stuff: the problem is regression testing, verifying that what you've just done hasn't broken what you did yesterday, or last week, or last month...One excellent way of minimizing the hassle is to create **command scripts**which you can replay to ensure that the whole game still hangs together after each set of changes.
Inform includes the verbs **RECORDING ON** and **RECORDING OFF**which create a script by capturing commands as you type them, and **REPLAY** which reruns (some or all of) a game by reading commands from a script file. These verbs are described in §7.1 of the Inform Designer's Manual (Edition 4/1).
RECORDING and **REPLAY** are available only when debugging is enabled, which isn't normally a problem while you're developing: Strict mode (and thence Debug mode) is the default compiler setting. However, when you release a game to beta testers, as a competition entry, or simply for general enjoyment, you'll want to turn off Debug mode -- by compiling with the **-~S** flag -- in order to prevent unscrupulous cheating. This means that these verbs now don't work; a bit annoying, since you'd really like to **REPLAY** your regression tests in regular mode as well as in Debug mode.
Here's the very simple way to make **RECORDING** and (especially) **REPLAY** available at all times. In addition, it also implements **!** as a comment verb, so that you can embed remarks in your script files to remind yourself of what's being tested. Just add this code to the end of your game, after the `Include "Grammar";` line:

```
#Ifndef DEBUG;
Global    xcommsdir;

[ CommandsOnSub;
    @output_stream 4; xcommsdir = 1; "[Command recording on.]";
    ];
[ CommandsOffSub;
    if (xcommsdir == 1) @output_stream -4;
    xcommsdir = 0; "[Command recording off.]";
    ];
[ CommandsReadSub;
```

```
    @input_stream 1; xcommsdir = 2; "[Replaying commands.]";
    ];

Verb meta 'recording'
    *       -> CommandsOn
    * 'on'  -> CommandsOn
    * 'off' -> CommandsOff;
Verb meta 'replay'
    *       -> CommandsRead;
#Endif;

[ CommentSub; ];
Verb meta '!'
    *       -> Comment
    * topic -> Comment;
```

In fact, you might find it convenient to place the code in a file `Replay.h`which you can then easily `Include` in all of your games. Finally, by way of example, here's a script file which exercises my example game "Cloak of Darkness":

```
! Command script to exercise "Cloak of Darkness" (Cloak.inf)
!
! NORTH goes nowhere
NORTH
!
! SOUTH is dark, and you need to come straight out again
SOUTH
NORTH
!
! Inspect your possessions
EXAMINE ME
INV
EXAMINE CLOAK
TAKE OFF CLOAK
DROP IT
!
! Investigate the Cloakroom
WEST
EXAMINE HOOK
TAKE IT
```

```
HANG CLOAK ON HOOK
TAKE CLOAK
HANG IT ON HOOK
EXAMINE HOOK
EAST
!
! Now OK to enter the Bar
SOUTH
TAKE MESSAGE
READ MESSAGE
```

Using the "system_file" and "replace" Directives In the Same Source File

here is an obscurity that presents itself when using both the replace and system_file directives in the same file.

First, to bring this article into context, let me describe briefly what they each do:

The system_file directive tells the compiler that the following routines, contained in the current source file, can be overridden via the replace directive. (It actually does more, but this is the behavior that is relevant)

The replace directive, generally called before the inclusion of the system_file, is used to indicate that certain routines are to be replaced. Specifically, when the replace directive is used to specify a routine name, instances of that routine, which occur under the scope of the system_file directive, are ignored. This way a new version, specified in a non-system file can be used instead.

The problem-behavior is that a file, which 1) specifies a function to be replaced, 2) implements a new version of the function, and 3) is designated as a system_file, has just specified its own implementation of the replaced function to be ignored as well. Compiling this file will result in an error message similar to:

```
testgame.inf(62): Error: There is no action routine called "ScoreSub"
```

For personal library extensions in particular, the need to replace standard library routines and provide new routines that can be replaced as well is common. What we seem to need, in cases like this is to specify some routines as replaceable and other routines as not replaceable.

The solution is rooted in the fact that the name of the system_file directive is potentially misleading since it seems to imply an entire file will be treated as a system file. Actually, this is not true. The manner in which the Inform compiler handles files designated as "system files" begins only when the directive is encountered and stops at the end of the file. This enables us to move the system_file directive down in our source file. The routines specified after the

system_file directive will be able to be replaced by games implementing the file, while the routines above the system_file directive will be compiled without the system_file qualifier and can therefore be used as replacements to library routines.

The following code snip demonstrates:

```
replace ScoreSub;

[ScoreSub;
   !...personal implementation of ScoreSub
];

system_file;

!...other code eligible to be replaced later goes here...
```

Part H
IF Theory

NPC Conversations: Ask/Tell Theory

1. Introduction

In the wake of such ground breaking games as "Galatia," "Photopia," and "Best of Three," NPC conversations have become all the rage. Although there are numerous ways to implement conversations (Emily Short identifies at least seven on her NPC Characterization page) the two dominant forms seem to boil down to conversation menus, and the age-old classic ASK/TELL.

As a developer I have always preferred the ASK/Tell system and most of the implementation of the ORLibrary has been based upon expanding this paradigm. This article, in fact, describes four different models for Ask/Tell implementation, details what purposes they serve in a story, and parallels their use in real life. It is also worth mentioning that, quite by coincidence, one of these models of the Ask/Tell theory lends itself particularly well to a menu interface.

This is an article on the theory of Ask/Tell, and I am attempting to keep away from platform specifics. For this reason, although library modules might be mentioned in passing, no code or implementation will be given here. For those who want more, however, this game is accompanied by a sample one-room conversation game, written in Inform, which bears some meager resemblance to both "Best of Three" and "Galatea." The code for this game, named "Medusa," will compile to both the Z-Machine and GLULX Native platforms. It leverages the ORLibrary framework, and can be found here:

http://www.onyxring.com/downloads/ORLib%20Examples/medusa.inf

A compiled version of the game (compiled to .Z5 format because it is more common right now than GLULX) can also be found here:

http://www.onyxring.com/downloads/ORLib%20Examples/medusa.z5

Additionally, to aid in understanding, a conversation map has been created and can be downloaded from the following URL:

http://www.onyxring.com/downloads/ORLib%20Examples/medusa%20conversatio
.jpg

For additional implementation information I would suggest the Advanced NPCs articles "Converstation and Learning" available from:

http://www.onyxring.com/InformGuide.aspx?article=20

2 Theory of the Ask/Tell Paradigm

How do we, as people, choose what we are going to say to another person? For the purposes of this article I will site four different forms of the human conversation model and relate these to topic based conversations. It is important to realize that each of these four forms of conversation is used in life to serve different needs just as each of these forms of conversation is used in a game to serve different needs.

2.1 The Librarian

The vast majority of IF games use this model. It is a response-only model where the NPC will respond to the player only if spoken to but otherwise does not initiate conversation. In life, this is a model we see in service staff, such as a librarian who does her duties and answers questions but otherwise minds her own business. The advantage of this model is that it is fairly easy to implement. This model has the disadvantage of only being player driven. Creative writing can temper the effect to some degree, but an NPC that simply answers questions isn't really much of a conversationalist.

2.2 The Rambling Idiot

In life, this conversation model is most often used by people who have just met. Particularly with strangers of whom nothing is known, there is little to do in a conversation than to make some random generic statement such as "How about them Yankees?" or "Nice weather we've been having." In a perfect world these little generic bits of conversation will lead to a more personal level of discussion ("The Yankees haven't played this bad since my freshman year of college." Or "Yes, this is great golfing weather!") but until the other person has "bitten," we are just fishing for a conversation.

In IF, the advantage of this model is that NPCs can initiate conversation on their own and do not have to wait for the player to address them. However, if used alone

this model has the disadvantage of not "flowing" from one topic to another. Normal conversation tends to progress in a chain of related subjects ("You followed the Yankees in college? What school did you go to?" or "You like to golf? So does my brother."). Without this relation between topics, the randomness of conversational blurbs makes people (or NPCs) appear to ramble as though intoxicated. This could be exactly what the developer is looking for in a character, but for a "normal" NPC, this model alone leaves something to be desired.

2.3 The Lecturing Professor

In life, we use this third form of conversation when we have something specific to say. A lecturer in a class room, for example, has specific points that must be made. As the lecture progresses, the points are made one-by-one until the lecture is concluded. It is possible that the students will keep quiet and the lecture will occur exactly as written on the lecturer's note cards. It is equally possible that members of the audience will ask questions.

When a lecturer answers a question, it can possibly be about related information that would not otherwise have been covered in the lecture. The question could also cover lecture material prematurely in which case those points need not be made again as the lecture continues.

In IF, this third model is ideal for ensuring that necessary information gets conveyed to the player. It is perhaps the most useful technique for writing NPC conversation "scenes" in a story since it conveys the appropriate information, yet allows the player to be as interactive (or not) as he/she desires.

One small disadvantage of this model is that the technique isn't truly "drivable" by the player. That is, there is a predetermined outcome that will come to pass regardless of what the player does. This is most likely what is desired in an IF story anyway, however this model simply doesn't work well for "conversation" games with multiple endings and "decision trees." For art to more closely imitate life, another model is needed...

2.4 The Conversationalist

This final conversation model is the logical continuation of the "Rambling Idiot" model discussed previously. In life, when we communicate with other people, we key into what is being said and keep a running pool of things to Ask/Tell which

are related to the conversation. Each statement can have several related topics which, in turn, have their own related topics. Together, these weave chains of interweaving topics (or a "Topic Web") which the player can navigate and be prompted with reasonable, natural statements/questions from the NPC. As an example, let's continue with one of the conversation statements covered previously:

```
"You like to golf?  So does my brother."
```

An NPC can be programmed to respond to this statement with a number of related statements, each response having its own set of related topics:

```
"Well, miniature golf. I worked at
'Golf World' after high school."
```

or

```
"Sure I like golf, but not as much as I
do water skiing.  I do that every
weekend."
```

or

```
"Not my brother, he hates golf with a
passion.  He got hit in the head with a
golf ball as a child."
```

or

```
"I think I know your brother.  Isn't
he on the debate team?"
```

Obviously the advantage to this system is that it allows for complex conversations seen in games such as "Best of Three" that can be driven equally by the NPC and the PC together. As an additional advantage, the concept of related topics is particularly well suited to support a menu. The menu, in turn addresses a disadvantage with the Ask/Tell paradigm as a whole: the difficulty of leading the player in the appropriate direction.

By its very nature, the Ask/Tell paradigm doesn't limit what the player can try to say. This openness of Ask/Tell is both a positive and a negative. It gives the player freedom since there need be no rhyme or reason to what is said to an NPC. If the player chooses, he can play the "rambling idiot" all he wants, but most players will look for clues when trying to determine what to say next. Without a menu, strong hints need to be made. One way to accomplish this would be through protagonist thoughts, but this technique grows more difficult as the number of conversation branches increases. For the four options above, several lines of text must be added:

```
"You like to golf?  So does my brother."

You begin to tell about your tenure at
"Golf World" but hesitate at the
mention of her brother whom you know from
debate.  He always reminded you of your own
brother who, oddly enough, hated golf.  You
two always had a great time skiing though.
```

As with any Ask/Tell system, this puts a fair burden on the shoulders of the author to make adequate suggestions about what the player might say next. Hopefully, with a bit of luck the player will read the above and consider one or more of the following:

```
>Tell girl about golf world
>Tell girl about brother
>Ask girl about brother
>Tell girl about skiing
```

Use of a menu system lightens this burden somewhat for both the author and player. This intertwining of menus and Ask/Tell has an additional advantage. Unlike non-Ask/Tell based menu systems, the player is not bound to choose one of these menu items. A menu used in conjunction with Ask/Tell can be exited and the player can "change the subject" by choosing another, non-related topic to Ask or Tell in the conventional manner.

The ease of a menu combined with the power of the command. Now that's flexibility!

3 Implementation

As I said up front, this article is about the theory behind Ask/Tell. There will be no code given here that describes implementation. Still, it would be irresponsible of me to simply end the article without some advice on where to go from here.

As a matter of personal preference, I am a big subscriber to the philosophy of reusable code. Most of the work that I do in this area I package into reusable modules and publish it in the ORLibrary. The sample, open source game produced for this article leverages the library and primarily demonstrates the fourth model discussed ("The Conversationalist") with a combined menu system. What follows is a list of pertinent ORLibrary modules that can be used to implement the models covered in this article:

3.1 The Librarian

ORKnowledgeTopic - This is the ORLibrary basis for all information used in the Ask/Tell paradigm as a whole.

ORNPC_AskTellLearn - NPCs derived from this class are given the ability to answer ASKed questions to which they know the answer.

3.2 The Rambling Idiot

ORNPC_Converse - This class instills NPCs with the ability to pick random subjects and TELL them to the player or another NPC.

3.3 The Lecturing Professor

ORKnowledgeScript - a derivation of the ORKnowledgeTopic class. This topic will interrupt the "Rambling Idiot" methodology and force the NPC to iterate through a list of topics (skipping topics that have already been talked about).

3.4 The Conversationalist

OROptionList - A class to support lists. This is not really an NPC component class, but NPCs can be derived from this to give them a "conversation pool" to retain topics for ASKing or TELLing.

ORKnowledgeWeb - a derivation of the ORKnowledgeTopic class. This object exposes a list of related topics that will be added to a characters "conversation pool" when told.