IAGE Developer's Manual

R.Rawson-Tetley

First Release (21st November 2001)

Table Of Contents:

1. What This Document Does and Does Not Contain

This document will teach you how to write single and multiplayer IF using the IAGE authoring system. However, it already assumes that you have some limited experience of the following:

1. Basic programming concepts (Variables, routines, arguments etc.)
2. Object oriented programming concepts (objects, classes and inheritance)
3. Interactive Fiction as a gaming genre

Note that I say “limited” - this document does not expect you to be an omnipotent code junkie and most concepts are explained as and when they are introduced, but some experience will be an advantage.

1.1 Starting Your Game

The IAGE compiler compiles normal text files into IAGE games. You will need to create an empty text file with the extension .ic to start with, using the editor of your choice. If you are using a Windows platform, I strongly recommend you get UltraEdit (www.ultraedit.com) as it is by far and away the best Win32 text editor in my experience, also, there is a special text file available on the IAGE site which allows UltraEdit to syntax highlight and autocomplete your code.

If you are not using a Windows platform, you will need a text editor which allows you to create ASCII text files in DOS format (CRLF line breaks).

1.2 Importing Libraries

The first thing you will need to do in your game file is import the IAGE core library - lib.ih. All IAGE library files should have the extension ih and should be stored in the same directory as the compiler (if you used the Windows distribution this will have been done for you).

To do this, we use the #import directive, like so:

```
#import “lib.ih”
```

When you import IAGE libraries, the compiler will first look in the same directory as the source file it is compiling (yourfile.ic) and if it cannot find it, it will then look in the compiler directory. This is useful if you want to make changes to the library for a game.

Although the compiler will allow you to put your imports anywhere in the source you like, it makes sense logically to put them at the very top.

1.3 Setting the Version

IAGE contains a set of library messages which you may override or add to yourself (IAGE can hold up to 2 billion library messages in theory depending on whether the machine can cope).

Library message 1 is always used to store the game’s name and version. Library messages can be over-ridden and created by using the LibraryMessage directive, followed by the message ID and text.

Eg:

```
LibraryMessage 1 "<b>~IF-Trek~</b><br><i>Interactive plagiarism by Bob</i>"
```

Note that the librarymessage directive (as with all textual statements outside of IAGE code) can be broken over multiple lines. When you do this, IAGE automatically throws away white space from the end of the first line and the beginning of subsequent lines, substituting them with a space. For example, in the message definition we just gave above, doing the following would produce the same result:

```
LibraryMessage 1 "<b>~IF-Trek~</b><br>
                  <i>Interactive plagiarism by Bob</i>"
```

Notice that speech marks appear around the string - these are IAGE's string delimiters and they are used to mark the beginning and end of a string. Also, notice the HTML tags embedded inside the text to print the name in bold and the headline in italics.

Make careful note of the <br> tag as you will need this a lot - this will insert a line break.

Because IAGE uses the speechmark as it's string delimiters, you are not permitted to use them inside strings. To obtain a speechmark, substitute the tilde character (~) instead. The compiler will convert them to speech marks when building the game file.

2. Creating Locations

Location objects are used to represent physical places, which players, items and NPCs inside your IAGE game can occupy.

To create a location, we use the Location directive, which accepts 2 arguments - one for the location's name in code and one for the display name, which is output to the players when they visit it. We use the Description statement to set the description the player sees when they are in this location.

To illustrate, we will add a location to the game we started above.

```
Location bridge "On the Enterprise Bridge" {
     Description "You stand at the front of the Enterprise bridge."
}
```

So now our game has one location object, which we can refer to as "bridge" in code, which will be called "On the Enterprise Bridge" when players visit it. It also has a basic description.

2.1 Lights

The IAGE library has lots of built-in routines for handling light and lightsources (which can get immensely complicated). Location objects can be flagged as having lights (for example from the electric overhead lights in the example above) or darkness.

If you do not say otherwise, IAGE assumes that your location has lighting and that players can see it. You may also specify that a location has lighting for clarity.

To do this, we use the "has" statement to assign this to the location.

Eg:

```
Location bridge "On the Enterprise Bridge" {
     Description "You stand at the front of the Enterprise bridge."
     has light
}
```

This is functionally identical to the previous example, since we did not specify whether the location had light or not, IAGE will assume that it does. If we replace the line "has light" with "has darkness", then IAGE will know that you want the location to be dark.

Notice at this point the curly brackets { } - these are used to mark where an object's definition begins and ends. You MUST supply these or the compiler will attempt to squeeze all your object definitions into one huge object!

The closing curly bracket must appear on the last line of the object definition ON ITS OWN. This is important, otherwise the compiler will not recognise it.

2.2 Scenery

While the location description we gave above for the bridge is functional (ie. It refers to everything of actual use in the location since there isn't anything yet), it is hardly a literary masterpiece, so let us rewrite it a bit.

```
Location bridge "On the Enterprise Bridge" {
     Description "You stand at the front of the Enterprise Bridge.
          Hi-tech computer consoles and busy crewmen surround
          you. Turbolifts lead away from the bridge to the
          northeast and northwest respectively."
      has light
}
```

Ok, so now it's a bit more interesting, except for the purpose of our game, none of those things that make the location more interesting are particularly relevant.

So what we can do is give IAGE a list of words that refer to those things in this location so if the player tries to do something with them, they are told that they are not important.

This is done with the Nouns statement:

```
Location bridge "On the Enterprise Bridge" {
     Nouns "computer" "consoles" "busy" "crewmen" "bridge" "turbolift"
          "turbolifts"
     Description "You stand at the front of the Enterprise Bridge.
          Hi-tech computer consoles and busy crewmen surround you.
          Turbolifts lead away from the bridge to the northeast and northwest
          respectively."
     has light
}
```

Any player, typing any of our listed nouns will receive the standard “That’s just scenery” message. If you do not want this message, you can create your own by using the Examine directive:

```
Location bridge "On the Enterprise Bridge" {
     Nouns "computer" "consoles" "busy" "crewmen" "bridge" "turbolift"
          "turbolifts"
     Examine "That does not need to be referred to."
     Description "You stand at the front of the Enterprise Bridge.
          Hi-tech computer consoles and busy crewmen surround you.
          Turbolifts lead away from the bridge to the northeast and northwest
          respectively."
     has light
}
```

What happens, is that the IAGE compiler generates a default item representing the location, contained inside it, which the player cannot see and this generates the message when it is referred to. If you are not happy with this functionality then you can implement your own by creating your own scenery objects (as you will see further on).

2.3 Exits

Of course, you will want to move around between locations. IAGE supports the standard 10 adventure directions (8 compass points, plus up and down), along with in and out for sublocations.

You specify these within your locations with the direction statements. We will create a new location at this point (since we will need to have somewhere to move to so the code will still compile).

```
Location bridge "On the Enterprise Bridge" {
     Nouns "computer" "consoles" "busy" "crewmen" "bridge" "turbolift"
          "turbolifts"
     Examine "That does not need to be referred to."
     Description "You stand at the front of the Enterprise Bridge.
          Hi-tech computer consoles and busy crewmen surround you.
          Turbolifts lead away from the bridge to the northeast and northwest
          respectively."
     NE_to turbolift1
     has light
}

Location turbolift1 "Turbolift 1" {
     Description "This is turbolift 1. The bridge is southwest."
     SW_to bridge
     has light
}
```

We can now move freely between the bridge and turbolift 1 by going northeast - notice the use of the NE_to statement. There is a statement for each compass point and up and down (U_to and D_to).

2.4 Non-Standard Exits

We don’t always want to just move players according to compass directions - exits can take some exotic forms - locations could be climbed upto, jumped down to, etc. etc.

To accomplish this, we will have to write some code now to cover this. Take for example the turbo lift exit above, we may want to let the player exit the bridge to the turbo lift by typing “ENTER TURBOLIFT” or “LEAVE BRIDGE”.

To do this, we need to trap a special location event - OnInput. This event fires every time a player makes some input inside a location.

We add an event trap to the location by using the OnInput directive. Code markers [] must surround where your code starts and finishes (note that the] closing marker must appear in a line ON ITS OWN) - eg:

```
Location bridge "On the Enterprise Bridge" {
     Nouns "computer" "consoles" "busy" "crewmen" "bridge" "turbolift"
          "turbolifts"
     Examine "That does not need to be referred to."
     Description "You stand at the front of the Enterprise Bridge.
          Hi-tech computer consoles and busy crewmen surround you.
          Turbolifts lead away from the bridge to the northeast and northwest
          respectively."
     NE_to turbolift1
     has light

     OnInput: [

          #enter,leave
               currentplayer.currentlocation = turbolift.id
               game.displaycurrentlocation
               end
          endif

     ]
}
```

So how did that work?

Firstly, the hash is a shorthand for testing what adverb a player typed. Note that when you test for an adverb, you may use ANY of it's synonyms in the condition (#in or #out would have worked equally well in place of #enter and #leave) and the comma represents a logical OR, so the condition to a human reads "If the player typed "enter" or "leave" (which is part of the adverb dictionary) then execute the following:".

The next line "currentplayer.currentlocation = turbolift.id" moves the player to the turbolift.

Currentplayer is an object, representing the player who entered the input we are testing. Currentlocation is a property of the player object (of which currentplayer is a subclass), which IAGE uses to track where the current player is by storing a location ID (we will see how sublocations are tracked later on).

Turbolift should need no introduction since it is the second location we defined. Since it is a descendent of the IAGE location object, it has an ID property which we can put in the currentlocation property to tell the player they are somewhere else now.

Game is an object containing useful library methods, of which displaycurrentlocation sends the current player's current location description to them.

The end command terminates code execution (our condition was satisified, so there is no point processing anything further).

The endif command is used to terminate the block of code being executed on successful matching of the IF condition (in this case the adverb matching of enter and leave). Notice how I tabbed the code so the inner code block which runs if the condition is met looks like it is "inside" the condition - this is good practice and you should follow it to ensure your code is legible.

Although it will be discussed in detail later, it is worth noting that IAGE events are carried out in a specific order and the act of displaying something to the player qualifies as a valid response and once an event has occurred which output something, no further events will be run (later, we will see how we can prevent this if this behaviour is undesirable).

2.5 Events

At the heart of IAGE is a powerful event system and all IAGE objects have their own events, which you can use to save time when building your game.

The location object supports two events:

2.5.1 OnInput

We have already seen the OnInput event in action in the previous example - this event is fired when a player makes some input in the current location. If you do not trap the input at this level, it “bubbles” up the event tree to further events.

2.5.2 OnDisplay

The OnDisplay event is fired immediately after the location is described to the player and before any objects, npcs or players are displayed. This event is useful for adding text to the end of the location description during play.

Eg:

```
Location bridge “On the Enterprise Bridge” {
     Nouns “computer” “consoles” “busy” “crewmen” “bridge” “turbolift”
          “turbolifts”
     Examine “That does not need to be referred to.”
     Description “You stand at the front of the Enterprise Bridge.
          Hi-tech computer consoles and busy crewmen surround you.
          Turbolifts lead away from the bridge to the northeast and northwest
          respectively.”
     NE_to turbolift1
     has light

     OnDisplay: [
          if ( currentplayer.getvalue(“uneasy”) = 0) then
               currentplayer.print “You have an uneasy feeling about all this.”
               currentplayer.setvalue(“uneasy”) = 1
               end
          endif

     ]

     OnInput: [

          #enter,leave
               currentplayer.currentlocation = turbolift.id
               game.displaycurrentlocation
               end
          endif

     ]
}
```

So now we have an OnDisplay event in our location. This event will add the message “You have an uneasy feeling about all this” at the end of the location description by outputting it to the current player (by calling the currentplayer object’s print method).

The player object allows you to attach custom name/value properties through the use of "getvalue" and "setvalue". If you attempt to access a name/value property that does not exist, it returns 0 as the answer. So the condition above says "Does this player have a value named "uneasy" with a value of 0?" and if it does, we print the message and then set the "uneasy" value to 1, so it will only be displayed the first time the location description is sent to the player.

So why did we store it with the player? All IAGE objects are capable of these getvalue/setvalue pairs, so why not store it with the location? Since we are writing a multiplayer game, if we stored it with the location only the first player who saw the location would receive the message (and we want all players to see the message).

2.6 Inline Events

As well as top-level events, IAGE supports a special kind of "inline" events. These allow you to define procedures inside object event code, which the server will "call back" when certain things happen. Whether you use them or not is entirely optional, but they offer some powerful facilities.

2.6.1 before_display

You may use this event to completely rewrite the location's description on the fly in code. The before_display event should be specified as a procedure declaration inside the OnDisplay event.

For example, if we wanted to alter the bridge's description to hide the crewmen, we could do something like this:

```
OnDisplay: [

     end
          proc before_display
               returnvalue = "You stand at the front of the Enterprise
                    Bridge. Hi-tech computer consoles surround you.
                    To the northeast is a turbolift."
               cancelevent
          end
]
```

In IAGE, you pass values out of procedures (notice that we declared it with the "proc" keywords and terminated it with the "end" keyword) by setting the returnvalue. In the case of the before_display event, the returnvalue should hold the new location description. Calling cancelevent tells IAGE that it should override the location description with the new one we have supplied. If we set the returnvalue and did not call cancel event, the original location description would be displayed.

This routine will therefore override the normal description so the player will never see the crewmen (or the northwest turbolift for that matter).

2.6.2 initialise

The initialise event is fired just once - when the game starts. This allows you to set any custom properties for the location and set up anything relating to the location. You may even add dictionary entries. The initialise event should be specified as a procedure declaration inside the OnInput event. Later examples display usage the initialise event.

2.6.3 each_turn

The each_turn event is fired every time a player makes a move (regardless of whether any of them are in this location or not). This is useful for location-based daemons. The each_turn event should be specified as a procedure declaration inside the OnInput event. Later examples display usage of the each_turn event.

3. Creating Items

Items make up the meat of your game world. Most of the IAGE library is dedicated to dealing with the manipulation of items. Items are basically used to represent every single non-animate object within your game world (tables, chairs, spaceships, buttons etc.)

We define items with the "Item" directive, which (like the location) accepts a code name and a display name. Let us add an item to our previous example.

```
Item cupoftea "a cup of Earl Grey tea" {
}
```

The code name and the display name are the barest essentials required to create an item. Also, notice that the display name includes the article (the letter "a"). Items should always include their article, as the IAGE library will make sense of everything.

As with the locations before, we should specify a list of nouns used to refer to this item. We also need a special statement which tells IAGE where the item should begin it's life (using the StartsIn statement).

Eg:

```
Item cupoftea "a cup of Earl Grey" {
     Nouns "cup" "earl" "grey" "tea"
     StartsIn bridge
}
```

When using the StartsIn statement, you do not have to just specify a location, you may also specify an another item (we will cover this further on in the lessons about containers).

There are also two special constant values you may use instead of a location, NPC or item name - limbo (the item starts nowhere) and random (IAGE generates a random location for the item to start in when the game begins).

In addition to these, items have a number of properties we can use as shorthand for generating messages. The two main ones are "Initial" and "Description". "Initial" contains the message displayed to players when the object is first encountered. The "Description" message is given when a player examines the item.

Eg:

```
Item cupoftea "a cup of Earl Grey" {
     Nouns "cup" "earl" "grey" "tea"
     Initial "On one of the computer consoles is a cup of Earl Grey tea."
     Description "The tea is fresh and steaming."
     StartsIn bridge
}
```

When using code, you can access these properties with (itemname).initialdescription and (itemname).defaultexamine. There is also another property "(itemname).movedfromoriginallocation" which is set to true once the item has been moved (and the initial description stops being used) - of course you can set or reset this value yourself to change the item's behaviour.

3.1 Portable Items

Unless you specify otherwise, all IAGE items are portable (ie. Can be picked up and carried around by players). It is possible to set a limit on the number of items a player can carry, also you may specify a limit on both the size and weight a player can carry. If you do not specify them, IAGE will set an item's Size and Weight properties to zero (in which case, you are limited to testing the number of items a player is carrying).

The item Size and Weight properties are completely arbitrary and you may use any scheme you wish (as long as they are whole numbers). For example, we could use weight units of say, quarter kilos and size units of quarter-feet cubed (to give size as a volume).

We could then set the size and weight of our cup of tea as 1 quarter kilo in weight and 1 quarter feet in size (probably not far off a cup of tea) - if this is not precise enough, simply invent your own scheme.

We set these properties using the Size and Weight statements, eg:

```
Item cupoftea "a cup of Earl Grey" {
     Nouns "cup" "earl" "grey" "tea"
     Size 1
     Weight 1
     StartsIn bridge
}
```

Note that when you are defining locations and items, it does not matter which order the statements appear in.

These properties can be got/set in code using (itemname).size and (itemname).weight.

3.2 Static Items

Of course, you don't want the player to be able to pick up and walk off with everything in your game (unless perhaps your subject matter is kleptomania), so you may flag items as "static". Here, we reintroduce the "has" statement, which is used to set object flags (recall the location had one where you could set light or darkness).

Static objects are also allowed a "static message", which is displayed to any player attempting to pick up a static item.

For example:

```
Item hugeboulder "a huge boulder" {
     Nouns "huge" "large" "boulder" "stone"
     StartsIn cavern
     StaticMessage "You are more like Clark Kent than Superman."
     has static
}
```

Would create a large boulder, which the player cannot move (with an almost humorous message). These properties can be accessed in code by using (itemname).isfixed and (itemname).fixedmessage. IsFixed will return true or false depending on whether the item is static or not.

3.3 Furniture

In addition to portable items, IAGE allows items to act as pieces of furniture, which players can sit on, stand on and lie down on.

These modifiers are set by using the sit, stand and lay flags in the has property. For example, we could create a sofa where the player can sit, stand and lie on it:

```
Item sofa “a leather sofa” {
     Nouns “leather” “sofa” “settee”
     StartsIn livingroom
     StaticMessage “Don’t be silly.”
     has static sit stand lay
}
```

We could prevent any of these (for example, create a chair which can only be sat on or stood on) by simply omitting one or more of the modifiers. Notice that the item still has to be flagged as static (a sofa is hardly very portable), although we could create a small stool that could be sat on that the player could carry around - simply by not making it static and giving it the sit modifier (has sit).

These properties can be accessed in code by using (itemname).canbesaton, (itemname).canbestoodon and (itemname).canbelaidon - all of which return true or false.

3.4 Openable Items

IAGE items can also be marked as openable (and indeed pre-open) by use of openable and open modifiers (has openable open). These properties can be accessed in code by using the (itemname).canopenclose and (itemname).openclosestate, both of which return true or false depending on whether the item can open and whether it is open.

For example:

```
Item brownbag “a brown leather satchel” {
     Nouns “brown” “leather” “satchel” “bag”
     has openable
}
```

Would create a leather bag item which could be opened and closed, and it would be initially closed. If we wanted it to be initially open, we add the “open” modifier:

```
     has openable open
```

Any item can be marked as open. There are a number of rules governing openable items when they are containers (which will be covered in the next section).

3.5 Containers

IAGE items may also act as containers for other items. You may “nest” items to an unlimited depth (up to virtual machine call stack space, which is a very large number indeed). We mark items as containers with the “container” modifier. This modifier can be accessed in code with (itemname).iscontainer.

For example, our leather bag example above could contain a wallet, which in turn could contain some money:

```
Location kitchen "Kitchen"
     Description "This is your kitchen."
}
Item brownbag "a brown leather satchel" {
     StartsIn kitchen
     Size 1
     Nouns "brown" "leather" "satchel" "bag"
     has openable container
}
Item wallet "a wallet" {
     StartsIn brownbag
     Size 1
     Nouns "wallet" "purse"
     has openable container
}
Item money "some money" {
     StartsIn wallet
     Nouns "money"
}
```

Notice how we are using the StartsIn property to pass an item instead of a location.

You can see from this how we are building an object tree like so:

```
Location -> Bag -> Wallet -> Money
           -> Player
```

When dealing with containers, the governing factor is size. The size property of all contained items must be less than the overall size of the container in the question. In our example above, all the containers have at least a size of 1 because all contained items default to a size of 0 (because we didn't specify it).

If we use the size scheme we outlined earlier of quarter feet volumes, then we could say:

brownbag size 4 (making the bag 1ft cubed)
wallet size 1 (making the wallet 1/4ft cubed)
money size 0 (it is smaller than unit of our scale)

This would track the sizes slightly more accurately.

There is an additional rule which applies to containers which are marked as openable. If a container is openable and the container is closed, the player cannot see what is inside the container unless you mark the container with the "transparent" modifier (accessed in code with (itemname).istransparent). Setting the transparent modifier allows the contents of a closed container to be seen.

3.6 Surfaces

In addition to containing other items, IAGE items can also have surfaces which other items can be put on. Similar to containers, the size property is used to determine just how many items can fit. We specify that an item has a surface by using the "supporter" modifier.

Eg:

```
Item table “the kitchen table” {
     Nouns “kitchen” “table”
     Size 24
     has supporter
}

Item sack “a brown sack” {
     Nouns “brown” “sack”
     StartsIn table
     Size 18
     has openable container
}
```

Would create a table we could put objects on upto a maximum of 6 square feet (using our previous size scheme). We can test whether an item has a surface in code with the (item).hassurface property.

In addition, a sack item container is created which uses StartsIn to be on the table. By adding the sit, stand and lie properties from the previous code examples to the table, players could also use it as a piece of furniture.

This now gives us a small problem - because the compiler can only recognise one item for the StartsIn property, what would happen if we specified an item has being both a supporter and a container? For example, a wardrobe can both contain things and have things on top of it.

Simple - if the compiler finds an item starts inside an item that is both a container and supporter, it puts the item inside the container. If the item is only a supporter, it puts the item on the container.

We can use a handy piece of code in the item’s initialise event to move the item to the surface when the game is started if we want an item to start on the surface of a supporting container. Eg:

```
Item wardrobe “a tall wardrobe”
     Nouns “wardrobe” “tall”
     has openable supporter container
}

Item suitcase “a battered suitcase”
     Nouns “battered” “suitcase” “case”
     has openable container

     OnAction: [
          end
          proc initialise
               this.currentlocation = wardrobe.surfacelocation
          end
     ]
}
```

This creates our supporter/container wardrobe and a suitcase. The suitcase takes advantage of the inline initialise event to move itself to the wardrobe’s surface when the game starts. The wardrobe can then be opened and closed, items can be put on top of and inside it when the game is running.

Notice the use of the special “this” object - the “this” object always refers to the current object the code is running in (in this case, the suitcase).

3.7 Weapons

IAGE features an inbuilt combat system and items can be marked as weapons. This is done by applying the “weapon” modifier (accessed using (itemname).isweapon). In addition, you can set the DamageIndicator property to determine the maximum hit points removed from NPCs/other players when using the weapon.

Eg:

```
Item sword “a sword” {
     Nouns “sword”
     DamageIndicator 10
     has weapon
}
```

Would create a sword, which can cause damage up to 10 hit points.

3.8 Lightsources

Items may also act as lightsources by using the “lightsource” modifier. This modifier automatically makes the item switchable and allows it to be switched off and on. In addition, the “light” modifier can be used to state whether the item is giving out light (you may manually set/remove this value if you do not want your lightsource to be switchable). These properties may be accessed in code with (itemname).islightsource and (itemname).islit.

Eg:

```
Item lantern “a brass lantern” {
     Nouns “brass” “lantern”
     has lightsource
}
```

Would create a lantern that could be switched on or off.

3.9 Scenery and Invisible Items

Where you refer to items in location descriptions, you can create an item to represent these things and mark them with the “scenery” or “invisible” modifiers (which are functionally identical). Setting these modifiers ensure that the item is not displayed to the player (because it is already mentioned in the location description).

Eg:

```
Location cargobay “Cargo Bay”
     Description “This is the cargo bay. Various boxes and canisters litter
                  the area.”
}
Item canister “Boxes and Canisters”
     Nouns “boxes” “canisters”
     has scenery
}
```

Creates a location with a scenery item that the player can refer to.

3.10 Subitems

Items may also act as subitems of other items. Subitems should be set as invisible and they act as components of their parent item and they are treated as part of the parent item.

We mark subitems with the "subitem" modifier and use the SubItemOf statement to tell IAGE what the subitem's parent is.

For example, suppose we want to create a device, which consists of a box with a large red button on the side.

```
Item device "The mystery device"
     Nouns "mystery" "device"
     Description "Little is known about the mystery device. There is a large red
                  button on the side."
}
Item redbutton "a large red button"
     Nouns "large" "red" "button"
     SubitemOf device
     has subitem invisible
}
```

This would create our device, with the button which could be referred to separately, and wherever the device goes, the button goes too.

3.11 Scenery Doors

Suddenly, we step into a slightly more complicated world. The IAGE library does not have anything "built" in for handling doors. The reason is that the IAGE library is built around the manipulation of individual items and doors are not individual items.

Yes, I know they are in other authoring systems, but they rely on trickery and having one door object effectively following the player between two locations. This is not acceptable behaviour for IAGE because a player could be stood in either location where the door appears.

What IAGE does supply is a small library you can import to deal with the handling of doors, however you are required to create two door items (one for each location the door appears in).

Here we introduce another IAGE concept - Code Modules. The IAGE library is a collection of code modules. Code Modules store a whole bunch of procedures, which you may call to perform tasks for you. You can even write your own code modules and procedures (which is a very sensible thing to do and we will cover this further on).

In the meantime, suppose we have two locations, which we want to have a door between. Say, a hallway and an office. We need to create the two locations to represent these places and two items - one for the door on each side.

Our first job is to import the door library. To do this, add this line to your list of imports at the top of your source:

```
#import "doors.ih"
```

We can now access the door functionality in our code.

Now, we need to create the locations and door items. I will demonstrate the entire code here for the working door system and then explain afterwards how it works:

```
Location hallway "Hallway"
     Description "This is a desolate hallway."
     E_to office
     OnInput: [
          ;go
               #e
                    if ( outerofficedoor.openclosestate = false ) then
                         currentplayer.print "The door is in the way."
                         end
                    endif
               endif
          endif
     ]
     OnDisplay: [
          call LockableTwoWayDoors.displaydoor outerofficedoor.id
     ]
}

Item outerofficedoor "the office door"
     StartsIn hallway
     Nouns "door" "office"
     has scenery openable
     Initial "To the east is the door to an office, which is open."
     ReadableText "To the east is the door to an office, which is closed."
     OnAction: [
          call LockableTwoWayDoors.doorhandler outerofficedoor.id innerofficedoor.id 0
     ]
}

Location office "Office"
      Description "A rather dull office."
      W_to hallway
      OnInput: [
          ;go
               #w
                    if ( outerofficedoor.openclosestate = false ) then
                         currentplayer.print "The door is in the way."
                         end
                    endif
               endif
          endif
     ]
     OnDisplay: [
          call LockableTwoWayDoors.displaydoor innerofficedoor.id
     ]
}

Item innerofficedoor "the hallway door"
     StartsIn office
     Nouns "door" "hallway"
     has scenery openable
     Initial "To the west is the door back to the hallway, which is open."
     ReadableText "To the west is the door back to the hallway, which is closed."
     OnAction: [
          call LockableTwoWayDoors.doorhandler outerofficedoor.id innerofficedoor.id 0
     ]
}
```

Most of this should make sense from what we have covered already.

You can see that we have two items now, representing the door in each location, and you can see that we have OnInput routines which check to see if the player is attempting to pass through the door while it is closed.

Each location has a “call LockableTwoWayDoors.displaydoor” line - call is the command used to run an external procedure. LockableTwoWayDoors is the name of the module the procedure resides in and displaydoor is the procedure we want to run. Anything after it are arguments to be passed to the procedure (in this case the ID of the door item we want to display).

Remember the earlier lesson about OnDisplay? This will ensure that the description of the door will be added to the location description. Because we are handling this description separately, we mark the door items as scenery (so their descriptions are not repeated).

If we set the open modifier on the items, the door would default to being open. The initial text contains the door’s description when it is open and the readable text contains the door’s description when it is closed.

The next item of interest is the OnAction event for items (which we have seen before) - this event fires whenever a player refers to that item in their input, and it occurs after the location.OnInput event has run.

In this event for the door items, we call another procedure in the door library - “doorhandler” and it requires three arguments - the ID’s of both door items and the third (which we have as zero in our example above), which is the ID of any key item for this door.

We could have set the door to be locked by adding a User Defined Boolean (see later section) flag named “locked” to the door items and creating a key item to open it with.

3.12 Enterable Items

In a similar fashion to the furniture items, mentioned above, items can be marked as enterable (using the “enterable” modifier). This property can be accessed in code using the (itemname).canbegotin property. Items marked as enterable allow the player to get inside them.

Apart from putting the player inside the item, the IAGE library does not do anything else. If you wish to display an extra description for inside the item, you can make use of some special properties.

We did not discuss it before with furniture, but when a player is “bonded” with an item in some way (in it, on it, etc.), two special properties of the player object are set - the “stateitem” property contains the ID of the item the player is affecting, and the “state” property contains a numeric constant, determining how the player is affecting that item. These constants are: 1 - No State, 2 - Sitting, 3 - Lying, 4 - Stood On, 5 - Inside.

For example, we could create a box that the player could get inside like so:

```
Item box “a large box” {
     Nouns “box” “large”
     Description “You may be able to get inside it.”
     has enterable
}
```

This is relatively simple stuff, and as stated before, as far as the library will help you. What if the box could be opened and closed? You wouldn’t want the player to be able to get inside the box if it was closed would you?

We can accomplish this behaviour with a little bit of code:

```
Item box "a large box" {
     Nouns "box" "large"
     Description "You may be able to get inside it."
     has enterable openable
     OnAction: [
          #in
               if ( this.openclosestate = false ) then
                    currentplayer.print "The box is closed."
                    end
               endif
          endif
     ]
}
```

Very simple - in the OnAction event (which fires when the item is referred to), we check to see if the player typed the adverb "in" (could be "get in box" or "enter box") and if they did, we check the box's open state. If it is closed, we give a message.

Because this message produces player output, the library doesn't bother trying to execute any further code, so the library GET IN/ENTER routines are never run (and the player doesn't get in the box).

I did mention earlier that we can stop this if we desire. Each player has a special property named "OutputToPlayer" which returns true or false, depending on whether the player has had anything output to them as a result of their last input.

By printing something to the player, and then adding the line "currentplayer.outputtoplayer = false" after them, we can reset this flag, so the library routines will still run and other events will fire.

Eg:

```
Item box "a large box" {
     Nouns "box" "large"
     Description "You may be able to get inside it."
     has enterable openable
     OnAction: [
          #in
               if ( this.openclosestate = false ) then
                    currentplayer.print "(Opening the box first)"
                    this.openclosestate = true
                    currentplayer.outputtoplayer = false
                    end
               endif
          endif
     ]
}
```

This changes our functionality somewhat - if the player attempts to get in the box now, we check the box's state (as before), and if we find it to be closed, we open it, tell the player they did it and then cancel the output flag so the library routine still runs to put the player inside the box.

3.13 Nearby Items and Scoping

Since location sizes are completely arbitrary and can represent an indeifinte amount of space, there are almost always going to be occasions where an item may be present, that you do not want the player to be able to actually reach (unless a puzzle is solved, or a stateitem is used to give them extra height etc.)

Part of the IAGE library is dedicated to scoping checks, preventing players from accessing items which are not in their stateitem - as long as you are using the library, you do not need to implement these kind of checks.

The second is a little more complicated - for example, you have an item on a high shelf, which you do not want the player to be able to reach.

To accomplish this, there is a module inside the IAGE library for coping with this, eg:

```
Item highshelf “a high shelf” {
     Nouns “shelf” “high”
     has supporter
}

Item ball “a beach ball” {
     StartsIn highshelf
     Nouns “beach” “ball”
     OnAction: [
          call NearbyItems.nearbyhandler this.id
          end
          proc before_get
               call NearbyItems.nearby_beforemove this.id
          end
          proc before_put
               call NearbyItems.nearby_beforemove this.id
          end
          proc before_remove
               call NearbyItems.nearby_beforemove this.id
          end
     ]
}
```

You can see that we make a call to NearbyItems.nearbyhandler with our item’s ID at the top of the OnAction event - this ensures that if the player attempts to do anything with the item, they will be told they cannot reach it.

The code below it traps all inline events for the item that allow it to be moved and ensure that it won’t be moved. This gives us a problem now, because the item will never be accessible!

The solution is simple - we only make the call to NearbyItems based on conditions, for example:

```
Item ball “a beach ball” {
     StartsIn highshelf
     Nouns “beach” “ball”
     OnAction: [
          if ( currentplayer.state <> 4 ) then
               call NearbyItems.nearbyhandler this.id
          endif
          end
          proc before_get
               if ( currentplayer.state <> 4 ) then
                    call NearbyItems.nearby_beforemove this.id
               endif
               end
          proc before_put
          if ( currentplayer.state <> 4 ) then
               call NearbyItems.nearby_beforemove this.id
          endif
          end
          proc before_remove
               if ( currentplayer.state <> 4 ) then
```

```
                    call NearbyItems.nearby_beforemove this.id
                endif
            end
        ]
}
```

This would allow the player access to the item if they were stood on something (their state is 4 - see previous lesson on enterable items).

Obviously, the condition is entirely upto the author (you could throw something at it, anything).

A better way of doing this would be to define a user defined boolean for the item, called "reachable" which is tested for determining whether the item is in scope or not. Then, depending on other actions, you could simply set or reset this flag at will. User defined booleans are covered in the next section.

3.14 User Defined Booleans

All IAGE items contain a special collection, named "User Defined Booleans" - these are flags you may attach to your items (there is no limit on how many per item).

You attach a user-defined boolean flag to objects with the HasUDB modifier, followed by the name(s) of your flags, separated by spaces. Note that you should only attach them with this modifier if you want the flag to be initially set - if you want it not set to start with, do not mention it at all (User Defined Booleans do not exist until you refer to them).

We test for the existence of the flags with (itemname).getuserboolean(boolname). This routine returns false if the item does not have the boolean flag, and true if it does.

Eg:

```
Item goldnugget "a shiny golden nugget" {
    Nouns "shiny" "golden" "nugget"
    HasUDB treasure
}
```

Would create us a golden nugget, with a special boolean flag of "treasure", which we could then test for with the (itemname).getuserboolean(treasure) flag. This gives us a very fast way of testing if a particular item is valuable or not.

User booleans may be added and removed from items in code using the (itemname).adduserboolean(name) and (itemname).removeuserboolean(name) methods.

In addition to these manipulation commands, there is also a player method named "getfirstboolitem" which will return you the first item a player is carrying that has the boolean flag you specify.

Using the example above, if a player was carrying the nugget, calling "currentplayer.getfirstboolitem(treasure)" would return the first item they had marked with the user boolean "treasure" - in this case goldnugget.id.

3.15 Custom Properties

In addition to the user-defined booleans, you can add custom properties to your items as name/value pairs. These custom properties can also be applied to players, npcs and locations.

They are stored as name value pairs with your objects and there is no limit on the number you may have on any given object.

There are no modifiers for these, and they must be retrieved/set in code using the (objectname).getvalue and (objectname).setvalue methods. If you attempt to retrieve the value of a property that does not exist, it returns a zero. If you want to assign initial values to a property, you should use the item/npc/location initialise inline events to assign a value.

Note that custom properties can hold numbers, strings or arrays.

Eg:

```
Item lantern “a brass lantern” {
     Nouns “brass” “lantern”
     has lightsource

     OnAction: [

          proc initialise
               this.setvalue(batterypower) = 100
          end
      ]
}
```

This would create us a brass lantern with a custom property value named “batterypower”, starting at 100.

You may find it confusing that the string names used in get/setvalue (and indeed in the user-defined boolean methods) do not require string delimiters - IAGE simply does not care, as it knows they must be string literals, so you are quite free to wrap string delimiters around them for your own clarity if you wish.

3.16 OnAction

As we have seen throughout this section, the item object has just the one event - OnAction, which is fired whenever a player refers to the object.

3.17 Inline Events

The item object, however, contains more inline events than any other object in the IAGE world:

initialise

The initialise event fires when the game is started, but before any players have joined. It is a handy event for setting initial property values and even adding dictionary entries.

each_turn

The each_turn event runs every time a player in the game makes some input (regardless of where the item is in relation to other players). This event is very useful for implementing turn-based daemons and timers.

before_get

The before_get event is fired by the library get routine, just before the item is taken by a player. If you wish to alter the message displayed to the player, set the returnvalue to your new message. If you wish to cancel the item being taken, call cancelevent.

Eg:

```
Item soap "a bar of slippery soap" {
     Nouns "bar" "slippery" "soap"
     OnAction: [
          end
          proc before_get
               returnvalue = "The slippery soap slips right through your fingers."
               cancelevent
          end
     ]
}
```

Would create a bar of soap that slipped through the player's fingers whenever they tried to pick it up.

before_drop

The before_drop event is fired by the library drop routine, just before the item is dropped by a player. As with before_get, if you wish to alter the message, set the returnvalue and call cancelevent if you want to cancel the drop.

before_insert

The before_insert event is fired by the library put routine, just before the item is inserted into a container. As with before_get, if you wish to alter the message, set the returnvalue and call cancelevent if you want to cancel the put.

before_remove

The before_remove event is fired by the library remove routine, just before the item is removed from a container. As with before_get, if you wish to alter the message, set the returnvalue and call cancelevent if you want to cancel the remove.

after_get, after_drop, after_insert, after_remove

These events are similar to their before equivalents, except they are fired after the event has taken place.

before_display

Similar to the inline location event before_display, the item before_display event allows you to change an item's name on the fly in code. Simply set the returnvalue to be the item's new name and call cancelevent to show your intent to override it. This event is fired whenever an item's name is referenced by the library for output, and it affects location descriptions and inventories.

before_displayinitial

This event is functionally identical to before_display, but only applies to the initial description given to an item before it is moved (so it only affects location descriptions).

3.19 Object Inheritance

Unlike every other kind of object in IAGE (where there is little point), items may inherit from other items. Inherited items take on the code of their superclass (which may be overridden), HOWEVER, the item properties are not inherited - any properties you wish to be inherited should be set using the item's initialise event in code.

Eg: (This is an example of the treasure class used in Zork).

```
Item treasure "Treasure Class" {

     OnAction: [

          end

          proc after_insert

               ' If player has inserted treasure item into
               ' trophy case and they have not received
               ' points for it, give them.

               if ( input.noun2 = trophycase.nounid ) then
                    if ( this.getuserboolean(hadpoints) = false ) then
                         this.adduserboolean(hadpoints)
                         call StandardLib.AddScore this.getvalue(points)
                         end
                    endif
                endif
          end

          proc before_get

               ' Points for finding treasure

               if ( this.getuserboolean(hadpickuppoints) = false ) then
                    this.adduserboolean(hadpickuppoints)
                    call StandardLib.AddScore this.getvalue(pickuppoints)
                    end
               endif
          end

          ' The initialise routine should be overridden in your subclass
          ' if you want to change the point value of a treasure item (just
          ' redefine the section below and change the 5 to something else).

          proc initialise
               this.setvalue(points) = 5
               this.setvalue(pickuppoints) = 2
          end
     ]
}
```

As you can see, the class handles awarding points for finding treasure and points for putting them in the trophy case in the living room. Here is an example of a subclass:

```
Item painting "a painting" extends treasure {
     Initial "On the far wall is a painting of unparalleled beauty."
     HasUDB treasure
     StartsIn studio
     Nouns "painting" "paint"

     OnAction: [

          ;smash,break,burn
               currentplayer.print "Don't be a vandal!"
          endif
     ]
}
```

As you can see, the "extends" keyword is used to show that an item inherits from another and it should appear after the displayname in the item definition.

Note (as the class definition shows), we could override the initialise event simply by redefining it in our subclass to set alternative values for the points awarded for collecting and storing this treasure.

The reason the treasure UDB is set is twofold - 1. An item is an item, regardless of inheriting from another, and 2. It's faster to test for a UDB than it would be to test for an instance of an item type.

The UDB definition is there for clarity - we could quite easily have added it to the initialise event to ensure that all treasures have it (this takes me back to the earlier point about using the initialise event for properties to extend).

4. Creating NPCs

NPCs are used to represent any kind of animate object within your games - ie. Anything which is "alive" in some way and can be spoken to. These objects can literally have a life of their own and obey logic. IAGE NPCs can also run in turn time or real time.

NPCs are created by using the "Character" directive. Like items, they have an initial description and an examine description. They also have a StartsIn property and collection of Nouns.

Eg:

```
Character picard "Captain Picard" {
     StartsIn bridge
     Nouns "captain" "picard" "jean-luc" "jean" "luc"
     Initial "Caption Picard looks pensively out of the window."
     Description "Captain Picard is the greatest of all the Star Trek captains"
}
```

Would create an NPC for Captain Picard.

4.1 NPC AI

Where NPCs differ radically from the other objects in the IAGE world is in their AI - IAGE supplies a number of AI states that you may use without the need for any coding. In addition, they also have a framework to assist you in writing your own AI routines.

NPCs have an "AIMode" property, which can only be set in code. This value can be one of 4 constants:

4.1.1 Dormant (0)

Whilst in this mode, the NPC does nothing. It just sits there and waits for instructions (custom author AI still runs in this mode, as it does in all modes).

4.1.2 Random (1)

In this mode, the NPC will walk around the map at random.

4.1.3 Following (2)

The NPC will follow the player with the Index specified in FollowPlayerIndex (this property can only be set in code at runtime).

4.1.4 Attacking (3)

In this mode, the NPC will attack the player with the Index specified in AttackingWho (this property can only be set in code at runtime).

If you set the modifier "autoattack" (same name in code), the NPC will attack the first player who enters the room automatically.

Setting the modifier "attackwhenattacked" (also same in code), the NPC will attack any player who attacks them first.

In addition, there is another property "DamageIndicator", which governs the maximum number of hitpoints the NPC can extract from a player in a single blow.

4.1.5 Pronouns

NPCs may be referred to with pronouns (him/her) - this functionality is automatically available inside the IAGE library, although it can be prevented by overriding the AfterInputImmediate event (discussed later).

To enable pronouns within your NPC, you will need to add a line to the top of the NPC's OnAction event:

```
OnAction: [
     call NPCPronouns.male this.id
]
```

This would enable an NPC to be referred to with the "him" noun - calling NPCPronouns.female instead allows use of the her noun.

4.2 Events

The NPC object has a number of events:

4.3 OnAction

As with the item OnAction event, this event is fired whenever the NPC is referred to by a player. This is why the pronoun code is placed in here (so the moment the NPC is referred to, we remember the last NPC the pronoun referred to).

4.4 OnTimer

The OnTimer event runs at a given interval of real time. The TimerInterval statement is used to set it and it's format is the number of milliseconds to run. For example, setting an NPC timer interval to 3000 would mean that it's OnTimer event was called every 3 seconds.

If you wish to use turn-based timers instead of real-time, there is a GameCode statement you can use - "RealTimeNPCs" - set this to no (the GameCode modifiers are discussed later).

Some words of warning about timer events:

Try to ensure that the code you write will run in much less time than the timerinterval is set for (otherwise the whole thing will grind to a halt) - this should not normally be an issue unless you are writing a very large routine to run in a very small timerinterval (try not to use intervals of less than 2 seconds).

Also, the currentplayer object is not valid in a timer event (because no user input spawned the code).

Mixing the AI Modes and Timers, we can do some interesting things, for example:

```
Character pig "a hungry pig" {
     Initial "A hungry pig snuffles and grunts here."
     Description "Skin hangs from the pig's bones - it is clearly starving."
     Nouns "hungry" "pig" "hog" "swine"
     StartsIn somelocation
     TimerInterval 4000

     OnTimer: [

          var i = 1
          while ( i < |item.count + 1| ) then
               if ( item(i).currentlocation = this.currentlocation ) then
                    if ( item(i).isedible = true ) then
                         item(i).currentlocation = 0
                         printallin this.currentlocation |"The pig eats
" & item(i).name & "."|
                         end
                    endif
               endif
               i = |i + 1|
          endwhile
     ]

     OnAction: [
          proc initialise
               this.AIMode = 1
          end
     ]
}
```

This creates a hungry pig NPC that wanders around the map randomly (we set it's AIMode to 1 in the inline initialise event) and when it finds an edible item, it eats it. Notice how we use the "printallin" statement to tell all players in the pig's location what it is doing. The while loop we created simply loops through all the items in the game and tests whether they are in the same location as the pig - if they are, they are then checked to see if they are edible.

4.5 OnTalk

The OnTalk event is unusual for an IAGE event, in that it is fired for two reasons - the first is because a player has attempted to address the NPC in the standard Infocom-style way "NPCName, <orders>". The input can be parsed as normal using the input objects verb/adverb/noun properties.

The second is that a player has used the "SAY" or "SHOUT" command to talk to other players in the room. NPCs can actually "hear" the said text and parse it themselves to determine if they want to respond.

The following example shows how this can be done using the input object (which is discussed in greater detail in a later chapter). In a more complex example later on, we will see how we can steer NPCs away from traditional stimulus/response programming, to create the illusion of intelligence.

```
Character conan "Conan the Barbarian" {
     StartsIn someloc
     Nouns "conan" "barbarian"
     Initial "Conan stands here, a picture of oiled biceps and gleaming teeth"
     DamageIndicator 50
     has attackwhenattacked

     OnTalk: [
            if ( input.isnpcaddress = false ) then

                if ( input.contains(fight) = true ) then
                    printallin this.currentlocation "Conan: I want fight!"
                    end
                endif

                if ( input.contains(sex) = true ) then
                    printallin this.currentlocation "Conan: I want sex!"
                    end
                endif
            else
                currentplayer.print "[ To address Conan, use ~SAY~ ]"
            endif
        ]
}
```

This creates us a barbarian NPC who can talk (extremely limitedly) about sex and fighting. This is a simple example, but I think it illustrates the point. Notice the first condition - we test the "isnpcaddress" property of the input object. This is set if the player typed "NPCName, <orders>" - since Conan is to behave like a human player, we don't bother with this form of address and tell the player so if they attempt to use it.

The input.contains method uses a pattern matching technique over the whole of the player's input before parsing, making it a very powerful tool for NPC interaction.

4.6 Inline Events

Every single NPC inline event should be specified in the OnAction event.

initialise

Functionally identical to the initialise event in other objects, this event fires when the game is first run and before any players have connected.

each_turn

Runs each time a player makes some input (useful for daemons).

before_display

Allows you to change the NPCs display name at runtime in code (set the returnvalue and call cancelevent, as with other versions of this event).

before_displayinitial

Allows you to change the NPCs initial description at runtime in code (set the returnvalue and call cancelevent, as with other versions of this event).

5. Multiplayer Games

5.1 Combat

IAGE features an inbuilt combat system that you may use in your games. The combat system is a simple "role-playing" type system.

We saw earlier in the NPC object how we can set "autoattack" and "attackwhenattacked" to govern NPC behaviour within the IAGE combat system. We also saw how NPCs have a DamageIndicator property, which determines how many hit points the NPC can take from a player when attacking them.

You can deactivate IAGE combat by setting the GameCode property "UsingIAGECombat" to no.

When IAGE combat is enabled, players may also attack each other (using attack <playername> [with] [weapon item]). The rules governing players are slightly more complicated.

Players have a DamageIndicator property (like NPCs), however this only applies to barefist fighting (not attacking a player WITH something). If the player uses a weapon, the DamageIndicator of the weapon is used instead.

The HitPoints property determines how many hit points a player has left before they will be killed, the ChanceOfHitting property is expressed as a percentage and contains the chance the player has of hitting someone when they attack them.

In addition, there are game properties, which you can use to set defaults for player combat:

DefaultDamage - contains the default DamageIndicator value for each player.
DefaultChanceOfHitting - contains the default ChanceOfHitting value for each player.
ChanceOfHittingIncrementForKill - contains the value to increase a player's chance of hitting by when they kill an NPC or other player.
DamageIndicatorIncrementForKill - contains the value to increase a player's DamageIndicator by when they kill an NPC or other player.

In addition, there are two properties governing death:

PlayersStayDead - contains yes or no, depending on whether players die when killed, or are respawned around the map at random with no items.
NPCsStayDead - contains yes or no, depending on whether NPCs die when killed, or are respawned around the map at random with no items.

Eg:

```
GameCode {
     DefaultDamage 5
     DefaultChanceOfHitting 30
     ChanceOfHittingIncrementForKill 5
     DamangeIndicatorIncrementForKill 2
     PlayersStayDead yes
     NPCsStayDead no
}
```

5.2 Money

IAGE also features a money system (if you wish to use it), activated by setting the GameCode property "UsingIAGEMoney" to yes.

Money is an abstract concept - it does not exist anywhere as a physical item, it is simply tracked as a quantity attached to the player (the player.money property).

You may assign a default money value to all players with the GameCode property "DefaultMoney".

It is left up to the author how they deal with transferring money when dealing with transactions/other players etc. However, if you are using IAGE combat as well, when you kill another player/NPC, you automatically receive any money they were carrying.

5.3 Actions with Multiple Players

From what we have covered so far, you should now have a pretty good idea of how the IAGE library works with regard to other players.

Every single player in the game is given a player object which you can access - there are three methods in which you can access player objects.

The CurrentPlayer object contains a reference to the player who sent some input that spawned the code now running. Two IAGE events will have an empty currentplayer object - OnTimer and OnInitialise - this is because these events do not run after any kind of player response.

The Player() collection holds all the players in the game, referenced by their index.

So:

```
player(currentplayer.index).print "Me!"
```

Would print "Me!" to the currentplayer, going via the player collection (although this is rather silly, since you could just use the currentplayer object).

Finally, the PlayerArray() collection holds all the players in the game, referenced by an indexed count. This collection makes it very easy to enumerate all the players in the game. The reason you use this instead of enumerating by index is that indexes are assigned dynamically and players can join and quit the game at any time, mean the indexes will not be sequential.

For Example:

```
     var i = 1
     while ( i < |player.count + 1|) then
          playerarray(i).print |"Hi there " & playerarray(i).name|
          i = |i + 1|
     endwhile
```

Enumerates all the players in the game and prints "Hi there <yourname>" to each one.

Since it would be a pain enumerating all the players in the game every time you wanted to show other players the consequences of one players actions, IAGE gives you a number of routines to do all this for you. In addition, all the library routines automatically tell other players of those particular actions (again, saving you a lot of time).

There are a number of printing functions that do not belong to any object, designed for printing to various players in various conditions.

PrintAllIn <location> <message>

Sends your message to every player in the location specified.

Eg:

```
printallin westofhouse.id |currentplayer.name & " opens the mailbox!"|
```

PrintAllInExcept <location> <playerindex> <message>

Sends your message to every player in the location specified, except for the player with the index you passed. This is useful if you want to display one message to the player doing something, and a different message for everyone watching, Eg:

```
PrintAllInExcept westofhouse.id currentplayer.index |currentplayer.name & "opens the
mailbox!"|
currentplayer.print "You open the mailbox."
```

PrintAllExcept <playerindex> <message>

Sends your message to every player in the game, except for the player with the index passed in.

Eg:

```
PrintAllExcept currentplayer.index |currentplayer.name & " smells!"|
```

Although it could perhaps be turned to a more adult use.

Right at the beginning, we discussed LibraryMessages. LibraryMessages are represented in the game with the message() collection. These objects also have a number of multiplayer methods:

Show outputs the library message to the current player.

ShowOthers outputs the library message to everyone in the current player's location except the current player.

ShowAllIn outputs the library message to everyone in the location specified except the current player.

The following code outputs the game version to the currentplayer, everyone in their location but them and finally to everyone in the currentplayer's location (excluding them)

```
message(1).show
message(1).showothers
message(1).showallin currentplayer.currentlocation
```

6. The GameCode Object

The GameCode object allows you to set parameters for your game. It also contains a number of game-specific events.

As with the other IAGE properties, you set them by specifying the propertyname, followed by a number, a delimited string, or yes/no.

Name (string)

Holds the name of the game - at present has no purpose, but I am to introduce software for IAGE communities and this will be required to inform players of the game an IAGE server is running.

MaxItemsCanCarry (number)

Determines the maximum number of items a player can carry.

MaxWeightCanCarry (number)

Determines the maximum weight of items a player can carry.

MaxSizeCanCarry (number)

Determines the maximum size of items a player can carry.

StartingLocation (locationobject name, limbo or random)

The location players will start in by default. You could move them yourself in the Start event, however this serves as a handy shortcut.

As mentioned before, limbo and random are valid locations, so you could elect for players to appear somewhere on the map at random.

Verbose (yes/no)

Determines if the game starts in verbose mode (descriptions are displayed regardless of whether a player has visited them before).

ShowAvailableExits (yes/no)

If set to yes, the library will automatically determine the exits from your locations and append them to the location descriptions for you.

I personally do not like this option, but it is handy for that MUD feel if you want to write a more traditional MUD.

AllowPersist (yes/no)

Determines whether players are allowed to save their position.

SinglePlayerGame (yes/no)

If set to yes, only one player is allowed to connect to the server, and when they save their position, the entire game state is saved (instead of just the player object).

MaxUsers (number)

The maximum number of users that may connect to a server running this game. There is no limit on the maximum amount of connections an IAGE server may service - the only limit is the internet connection and machine memory.

RealTimeNPCs (yes/no)

If set to no, NPC AI and events run at the end of each player's turn instead of in real time intervals.

WideInventoryDisplay (yes/no)

Determines whether location/player inventories are shown in wide or tall format.

MediaBase (string)

Optional value - if you choose to use this, set it to a URL base, which IAGE will prefix all media names with.

OverrideSecondaryNouns (string)

This property should include a list of verb IDs - if IAGE encounters one of these verbs, every noun after the first will not be checked for availability before continuing.

By default, these library sets this to “66 68 69” - these are the IDs of verbs “ask”, “tell” and “consult”. This allows players to refer to objects which aren’t in scope when dealing with these actions.

For example, “ASK RAYMOND ABOUT THE SHOE” - “raymond” will be checked for availability (we are asking him), but the shoe won’t.

6.2 Overriding and Extending

When dealing with IAGE properties, they are overridden - the library specifies defaults for all the GameCode properties and setting them to something different in your game overrides the original setting.

The GameCode events can be extended, as well as overridden (as with item inheritance).

For example, suppose the library defines the game event OnStart, like this:

```
GameCode {

      Start: [
            game.displayversion
            game.displaycurrentlocation
      ]
}
```

You could provide your own implementation by redefining it in your code with the “Override” operator:

```
GameCode {

      Start: Override [
            game.displaycurrentlocation
      ]
}
```

This would stop the game version being output on startup.

We could also extend the functionality of the original routine by using the “CodeExtend” operator. This causes the new code to be added to the original.

Eg:

```
GameCode {
      Start: CodeExtend [
            currentplayer.print “Welcome!”
      ]
}
```

Would add the message “Welcome!” to the code, causing it to be output to new players on startup.

6.3 Game Events

The GameCode object has a number of events:

Initialise

Runs when the game starts up, but before any players have entered.

Start

Runs whenever a new player joins the game.

Quit

Runs just before a player's connection is closed.

DisplayBanner

Returns the value to display in the banner.

Score

Outputs the player's score to them.

AfterInputImmediate

Runs immediately after a player makes some input.

AfterInput

Runs after all other events have been checked as a result of a player's input.

BeforeInput

Runs just before control is returned back to the player for more input.

6.4 Event Tree

Now that we have mentioned all the IAGE events, we should look at putting them into some sort of coherent order. When a player makes some input, the events are fired in the following order:

All game items are enumerated and their each_turn inline event called.
All locations are enumerated and their each_turn inline event called.
All characters are enumerated and their each_turn inline event called.
The Game's AfterInputImmediate event is called.
If it produced no player output, the OnInput event of the location the player is in is called.
If no output has been produced, the OnAction event of every item the player referred to is called.
If no output has been produced and the player was addressing an NPC, the OnTalk event of it is called.
If no output has been produced, then the Game's AfterInput event is called..
If we still have nothing to send to the player, the IAGE library is called.
Finally, regardless of any output now, the Game's BeforeInput event is called and control returns to the player.

7. Libraries

7.1 Victory and Death

The IAGE library contains two standard routines for victory and death - Won and Dead, as part of the StandardLib module.

Eg:

```
Location nasty "Horribly Unfair" {
      Description "This is a nasty room. Two doors lead ead and west - one to death,
one to victory, but which one?"

      OnInput: [
            ;go
                  #e
                        call StandardLib.Dead
                        end
                  endif
                  #w
                        call StandardLib.Won
                        end
                  endif
            endif
      ]
}
```

Would create a room that either killed a player or let them win depending on which direction they went.

7.2 Scoring Points

IAGE does not take much of an interest in player scoring, and you are generally left to implement your own scoring systems.

The library tracks player's scores and the default routines display score and turns. There is also a library routine "AddScore <amount>", which you should use to give points to players, since it generates the relevant notifications.

Eg:

```
call StandardLib.AddScore 5
```

Would give the current player five points, and produce the output "[Your score just went up by 5 points]".

7.3 Switchable Items

While IAGE does not supply native support for switchable items (excepting lightsources), there is an additional library you may import, named "switches.ih". You will need to import this in your code, using the #import directive.

This library allows for four different types of switches:

On/Off Switches

On/Off switches have to be explicitly switch on and off by the player "Switch x On" or "Switch On x".

To use on/off switches, set the user boolean "ison" if you wish the item to be on by default. Simply add the line “call SwitchableItems.switchhandler this.id” in the item’s OnAction event.

Toggle Switches

Toggle switches can just be switched and they toggle between on/off states.

To use toggle switches, set the user boolean “ison” if you wish the item to be on by default. Simply add the line “call SwitchableItems.toggleswitchhandler this.id” in the item’s OnAction event.

Toggle Levers

Toggle levers work just like toggle switches (except they must be pulled).

The only difference is the call; “call SwitchableItems.toggleleverhandler this.id”

Up/Down Levers

Up/Down levers have to be explicitly pushed up and down by the player “PUSH X UP”

To use them, set the userboolean “isdown” if you want the item to be in the down position by default. Then add the line “call SwitchableItems.leverhandler this.id” to your item’s OnAction event.

8. Parsing

8.1 Parser Breakdown

8.2 The Input Object

The input object is the author’s only method of deciphering what the player typed, and it contains a number of useful properties and methods:

verb, verb2 and verb3

Returns the ID of the verbs typed in the sentence in the order they were typed (verb appears before verb2 and verb3).

adverb, adverb2 and adverb3

Returns the ID of the adverbs typed in the sentence in the order they were typed.

noun, noun2 and noun3

Returns the ID of the nouns typed in the sentence in the order they were typed.

contains(text)

Returns true if the string fragement in “text” appears anywhere in the player’s input.

isnpcaddress

Returns true if the player was addressing an NPC with the input (NPCName), (command)

totalwords

Returns the number of words the player typed.

word(index)

Returns the word with the index passed in that the player typed.