TinTin++ Unofficial Documentation Documentation

Release 0.1

Nathan Farrar

September 02, 2015

Contents

CHAPTER 1

About

This is an unofficial documentation resource for the TinTin++ client (hopefully a community effort). The documentation for TinTin++ is not automatically built and is severely out of date. Because user's (including myself) are constantly asking the same questions on the tintin forums, I have attempted to document those features that cause the most issues here and lay the documentation out such that the starting sections begin with the questions and issues encountered by the newest users and the later sections are those encountered by more advanced users.

The cookbook exists outside of this layout and consists of 'shorter' recipes to perform very specific tasks that are commonly inquired about on the forums.

CHAPTER 2

Features

Supplemental documentation for things that are either not obvious or undocumented.

2.1 Sessions

2.1.1 The Startup Session

The startup session is created at default. It's named gts (per #help session) and can be referenced with #gts, just as you would reference a character's session. Any code from command files loaded on startup is executed in this session.

There are probably other interesting things about #gts, but I don't know them.

2.1.2 Session Inheritance

When creating a new session - the entire config from the startup session is copied into the new session. This is important if you play multiple sessions at the same time - you don't want to copy character A's data into a session for character B.

It is also important to note that the copy of the data does not reference the original session in any way - it's a brand new copy. Once the new session has been established, changing the data in either session will not affect the copy of the data in the other.

If you use a "session API" (ttlib includes one), then this can be confusing at first. When I load the program, all my session management code is copied into the GTS session. When I create a new session, that code gets copied into the characters session. I can then use the session API from the characters session, however it will contain different variables than those in the #GTS session.

2.1.3 References

- The All Command
- The Session Command

2.2 Data Types

There are several different data types in tintin, several of which are undocumented. They must be used in specific ways that vary by context, so it can get tricky:

- Simple variables: The basic building block for all data types.
- String lists: A simple variable, used in the context of a foreach loop that contains semicolon delimited fields.
- Brace lists: A brace delimited type that is only valid in the context of a foreach loop input.
- Nested variables: A variable containing another variable. Try not to think of this as arrays, this will lead you to dark places.
- Lists: A complex data type that uses integers for it's indexes. An API is provided for working with this data type.
- Tables: A complex data type consisting of name/value pairs.

2.2.1 Simple Variables

A simple variable:

```
#var SIMPLE_VARIABLE value;
```

These are self explanatory, enough said.

2.2.2 Removing Variables

To remove a variable, use #unvariable (or #unvar). When planning out your data storage, keep in mind that you can deeply nest variables & tables in the same "root" variable, then remove the entire thing at once, using #unvar nested_data;

You cannot remove multiple variables at once, i.e:

```
#unvar var1 var2 var3;
```

You must remove them individually:

```
#unvar var1;
#unvar var2;
#unvar var3;
```

This is worth considering when you design your data structures.

2.2.3 String Lists

A string list is a simple variable that contains a string with semicolons that can be interpretted in the context of a foreach loop. From #help foreach:

```
#foreach {bob;tim;kim} {name} {tell $name Hello}
```

2.2.4 Brace Lists

Brace lists confused me pretty bad. They are only valid in the context of a forloop and cannot be stored in a variable. Again, from #help foreach:

```
#foreach {{bob}{tim}{kim}} {name} {tell $name Hello}
```

The format for these lists is extremely similar to that of a "table" (more on tables later).

If you attempt to use the brace list syntax outside of the context of a forloop input, you'll get confusing behavior:

```
#var list {{bob}{tim}{kim}};
#OK. VARIABLE {list} HAS BEEN SET TO {{bob}{tim}{kim}{}}.
```

The parser created the "list" but added an extra {} at the end. This is because the parser is interpreting the input as a table, and tables store key/value pairs. So to recap - you cannot create "brace lists" outside of the scope of a #foreach input.

2.2.5 Nested Variables

Variables can be nested using either "array" or "dot" syntax. Both of the following forms are syntactically correct:

```
#var {V1.N1.N2}   {VALUE};
> #VARIABLE {V1}={{N1}{{N2}{VALUE}}}

#var {V2[N1][N2]} {VALUE};
> #VARIABLE {V2}={{N1}{{N2}{VALUE}}}
```

Note the syntax, it's important. The exploded representation:

```
#VAR V1
{
    {N1}
    {
        {N2}{VALUE}
    }
}
```

Each "nested" variable is encapsulated with an extra set of braces. Note: I've run into some situations where whitespace in nested variables and tables matter, so don't define them in your code using the exploded form above (no extra whitespace!).

You can retrieve the values from nested lists using either syntax:

```
#showme $V1.N1.N2;
> VALUE

#showme $V1[N1][N2];
> VALUE
```

And in some cases (though I don't know why you'd want to) you can mix the "array" and "dot" syntax:

```
#nop This works.;
#showme $V1.N1[N2];
> VALUE

#nop This does not work.;
#showme $V1[N1].N2};
> {N2}{VALUE}.N2};
```

2.2.6 Lists

Lists are essentially integer indexed arrays:

```
#list list add item1;
#list list add item2;
#list list add item3;
```

```
#list list add item4;
> #VARIABLE {list}={{1}{item1}{2}{item2}{3}{item3}{4}{item4}}
```

Note that the order is preserved in a list:

```
#list list add item0;
> #VARIABLE {list}={{1}{item1}{2}{item2}{3}{item3}{4}{item4}{5}{item0}}
```

You can add multiple values to a list at once, note that they are whitespace delimited:

```
#list list add item1 item2 item3 item4;
> #VARIABLE {list}={{1}{item1}{2}{item2}{3}{item3}{4}{item4}}
```

Like (just about) everything in tintin, you can disable whitespace delimiting by using explicit braces:

```
#list list add {this is a sentance}
> #VARIABLE {list}={{1}{this is a sentance}}
```

NOTE: I haven't used these types of lists as much as nested variables and tables, so I'm unsure if forcing whitespace into items will break the API. For more information about lists, see #help list.

Tables

Tables are key/value pairs, stored in variables. They are undocumented, but very powerful:

```
#var {T1} {{K1}{V1}{K2}{V2}};
> #VARIABLE {T1}={{K1}{V1}{K2}{V2}}
```

Notice that the parser writes these differently than it writes nested variables. They are different, and handled differently. They are however stored exactly the same as lists, but with string-based indexes, rather than integers.

Tables are automagically sorted. If you need to preserve the order, you'll have to use another data type. AFAIK, there's no way around this:

```
#var {T1} {{K2}{V2}{K1}{V1}};
> #VARIABLE {T1}={{K1}{V1}{K2}{V2}}
```

Values can be inserted into tables using either "array" or "dot" notation:

```
#var T1[K1] V1;
> #VARIABLE {T1}={{K1}{V1}}

#var T1.K1 V1;
> #VARIABLE {T1}={{K1}{V1}}
```

This is exactly the same syntax as the "nested" variables referenced above. The only difference is that we can shove multiple key/value pairs into a single variable:

```
#var T1.K1 V1;
#var T1.K2 V2;
> #VARIABLE {T1}={{K1}{V1}{K2}{V2}}
```

Note: While the data in the variable looks like a "brace delimited list", it is not. Don't get confused on this, it caused me great trouble. This form is ONLY used for storing key value pairs.

When referencing a table stored in a variable, we can pull out a list of the indexes using a special "[]" operator that's only valid for this purpose in this context:

```
#showme $T1[];
> {K1}{K2}
```

We can store tables in any value inside a nested variable:

```
#var V1.N1.T1 {{K1}{V1}{K2}{V2}};
> #VARIABLE {V1}={{N1}{{T1}{{K1}{V1}{K2}{V2}}}}
```

And retrieve it:

```
#showme $V1.N1.T1;
> {K1}{V1}{K2}{V2}
```

To iterate over the values, we have to be careful:

```
#foreach {$V1.N1.T1} {table} {
    #showme $table;
}
> K1
> V1
> K2
> V2
```

Referencing it this way in the #foreach loop input causes the parser to interpret the table as a brace list (remember above?). Not the behavior we want.

To iterate over the table correctly, we need to use the special "[]" operator mentioned above:

```
#foreach {$V1.N1.T1[]} {table} {
    #showme $table;
}
> K1
> K2
```

We can store multiple tables in a nested variable:

```
#var V1.N1.T1 {{T1K1}{T1V1}{T1K2}{T1V2}};
#var V1.N1.T2 {{T2K1}{T2V1}{T2K2}{T2V2}};
> #VARIABLE {V1}={{N1}{{T1}{{T1K1}{T1V1}{T1K2}{T1V2}}{T2}{{T2K1}{T2V1}{T2K2}{T2V2}}}}
```

To get a list of our tables stored in N1, we can use:: #showme $V1.N1[]; > {T1}{T2}

However, we have to be careful how we attempt to iterate over them. The following gives us the name of each table, a newline, and then the table content:

```
#foreach {$V1.N1} {table} {
    #showme $table;
}
> T1
> {T1K1}{T1V1}{T1K2}{T1V2}
> T2
> {T2K1}{T2V1}{T2K2}{T2V2}
```

To just get the name, we have to use our "[]" operator:

```
#foreach {$V1.N1[]} {table} {
    #showme $table;
}
> T1
> T2
```

As far as I know, there is no way to pull the data for each table directly out. This requires a little more effort, though we can use the following syntax to accomplish the task:

```
#foreach {$V1.N1[]} {table} {
    #foreach {$V1.N1[$table][]} {key} {
        #showme $table:$key:$V1.N1[$table][$key]
    }
}
> T1:T1K1:T1V1
> T1:T1K2:T1V2
> T2:T2K1:T2V1
> T2:T2K2:T2V2
```

The syntax for the foreach input fields above is extremely picky. I played around with several variations using dots and braces, and this was the one I found that works.

2.2.7 References

- The Variable Command
- The List Command
- Retrieving Keys from an associative lists
- Working with associative lists
- Stumped Nested List
- Forcing Variable Substitution
- Help with List Issues

2.2.8 Notes

TODO: Add information about checking for variable and index existence (&<variable name>). TODO: Add notes on escaping periods in variable names (#var LOGFILE data/log/$ISODATE.log;).

2.3 Command Files

Commands can be placed in files (known as command files).

2.3.1 Reading Command Files

Reading and then read into TinTin using two basic methods:

```
#class {<classname>} {read} {<filename>}
#read {filename}
```

The #class {read} {filename} command is basically shorthand for:

> #class {name} {open}; #read {filename}; #class {close};

Using the #class {read} {filename} method imposes some limitations on us:

1. Everything in the file will be part of the class (we can't mix non-class code in a file read in this way).
2. We don't explicitly remove the existing data before reading it in, therefore unexpected results can occur (if reloading command files on the fly).

I prefer the more flexible approach (using the #read command with explicit #class commands) as described next.

2.3.2 Execution Order

Command files can load each other. When they do, the command file is immediately read and the commands are immediately executed as soon as #read statement is encountered. The following scripts demonstrate this behavior:

```
#nop init.tin;

#showme {<fba>DEBUG:INIT         Start of init.tin.};

#alias read {
    #showme {<fba>DEBUG:INIT:READ    Reading %1.};
    #line verbose #read %1;
}

#showme {<fba>DEBUG:INIT         Just before read alias commands.};

read module1.tin;
read module2.tin;

#showme {<fba>DEBUG:INIT         Just after read alias commands.};

#showme {<fba>DEBUG:INIT         Reading module3.tin started.};
#read module3.tin;
#showme {<fba>DEBUG:INIT         Reading module3.tin completed.};


#nop module1.tin;
#showme {<fba>DEBUG:MODULE1:     Inside of module1.tin.};

#nop module2.tin;
#showme {<fba>DEBUG:MODULE2:     Inside of module2.tin.};

#nop module3.tin;
#showme {<fba>DEBUG:MODULE3:     Inside of module3.tin.};
```

When we execute init.tin with:

```
tt++ -v -r init.tin
```

We see the following output:

```
#CONFIG {VERBOSE} HAS BEEN SET TO {ON}.
#CONFIG {TINTIN CHAR} HAS BEEN SET TO {#}.
DEBUG:INIT         Start of init.tin.
#CONFIG {COMMAND ECHO} HAS BEEN SET TO {ON}.
#CONFIG {VERBOSE} HAS BEEN SET TO {ON}.
#OK. {reload} NOW ALIASES {#system {tput clear};#kill {all};#read init.tin;} @ {5}.
#OK. {read} NOW ALIASES {#showme {<fba>DEBUG:INIT:READ    Reading %1.};#line verbose #read %1;} @ {5
DEBUG:INIT         Just before read alias commands.
DEBUG:INIT:READ    Reading module1.tin.
#CONFIG {TINTIN CHAR} HAS BEEN SET TO {#}.
DEBUG:MODULE1: Inside of module1.tin.
DEBUG:INIT:READ    Reading module2.tin.
#CONFIG {TINTIN CHAR} HAS BEEN SET TO {#}.
DEBUG:MODULE2: Inside of module2.tin.
DEBUG:INIT         Just after read alias commands.
DEBUG:INIT         Reading module3.tin started.
#CONFIG {TINTIN CHAR} HAS BEEN SET TO {#}.
DEBUG:MODULE3: Inside of module3.tin.
```

```
DEBUG:INIT          Reading module3.tin completed.
```

2.4 Loops

There are six commands in tintin for implementing different sorts of "loop" behavior:

1. Foreach: For each item in the specified list, retrieve the value and execute some command(s).
2. Forall: For all items in the specified list, execute some command. This command appears to be identical (except for the "goblin" syntax) to the #foreach command.
3. Loop: Loop from and to specified indexes (a traditional for loop).
4. While: Execute a set of command(s) until a specified condition is met.
5. Parse: Iterate over a string and for each character, execute some command.
6. Repeat: Execute some command a specified amount of times.

There are also two flow-control commands that can be used to alter the flow of the loops (though they appear to have identical behavior):

1. Break: Stop executing the current loop and resume execution at the end of the loop's control structure.
2. Continue: Stop executing the current loop and resume execution at the end of the loop's control structure.

2.4.1 Goblins

I honestly can't think of a better name "looping through lists" in TinTin. The syntax is completely nonstandard and doesn't fit into the same context as anything else in TinTin++. I can never remember the syntax and have to look it up each time I write one.

From the foreach page of the official manual: > To loop through all items in a list (or a nested variable) using the foreach command use $<list>[%*].

The forall loop uses a special "goblin" syntax for referencing the value of the current item: "&0".

Examples

Example syntax:

```
#foreach {$mylist[%*]} {value} {
    #showme {$value};
};

#forall {$mylist[%*]} {
    #showme &0;
};

#loop {1} {5} {i} {
    #showme {$mylist[$i]};
};

#var {i} {5};
#while {$i > 0} {
    #showme {$mylist[$i]};
    #math {cnt} {$cnt - 1};
};

#parse {a string of characters} {c} {#showme $c};

 #10 #showme {repeat};
```

References

- The Break Command
- The Continue Command
- The Forall Command
- The Foreach Command
- The Loop Command
- The Parse Command
- The Repeat Command
- The While Command
- Stumped on Nested List
- Retrieving keys from associative array variables
- Iterating Lists with foreach
- how do same thing like ismember

2.5 Colors

NOTE: This page is incomplete.

References: - RGB Colour Conversion - Colour Questions

2.6 Tab Completion

Tab completion does not work for commands with whitespace, the following script demonstrates this behavior:

```
#nop Our "nested" tab completions.;
#tab module list;
#tab module load;
#tab module kill;
#tab module reload;

#nop Our primary API alias. Calls the correct aliases for us based on the ;
#nop initial arguments;
#alias module {

    #showme ALIAS: module;

    #if {"%1" == "list"} {moduleList};
    #elseif {"%1" == "load"} {moduleLoad %2};
    #elseif {"%1" == "kill"} {moduleKill %2};
    #elseif {"%1" == "reload"} {mdouleReload %2};
    #else {
        #showme ERROR: Usage: module list | load <module> | kill <module> | reload <module>;
    };
}

#nop Some stub aliases.;
#alias moduleList   {#showme moduleList};
#alias moduleLoad   {#showme moduleLoad %1};
#alias moduleKill   {#showme moduleKill %1};
#alias moduleReload {#showme moduleReload %1};
```

This results in the following tab completions:

```
mod<tab>
module kill<tab>
module mod<tab>
module module kill<tab>
module module mod
... etc
```

2.7 Shell Integration

There are several commands provided in TinTin to facilitate shell integration:

- #script {variable} {shell command}
- #run {session name} {shell command}
- #system {command name}

2.7.1 Notes

NOTE: This page is incomplete. TODO: Add detailed notes about #script (out of these commands I understand it the best).

2.8 Mapper

TinTin's mapper is the best available. It's the reason why TinTin is so popular, even after all these years.

2.8.1 Getting Started

Creating a Map

To create a map, use:

```
#map create <size>
```

The size argument specifies the maximum number of rooms the map can hold (this can be increased later, if necessary). I do not believe the maximum number of rooms affects the map performance (though the actual number of rooms absolutely does), so I typically create my maps with a very large number of maximum rooms (1,000,000). If no size is specified, the default value of 50,000 is used.

Modifying the Maximum Room Count

To modify the maximum number of rooms, use:

```
#map resize <number>
```

I haven't ever had to do this, because I always create my maps with a starting size of 1,000,000 rooms (and I haven't gone over this number). As mentioned above, the maximum number of rooms does not affect map performance, so there are no drawbacks to doing this.

Destroying a Map

A map that is loaded can be destroyed with the command:

```
#map destroy
```

This will remove it from memory and make us leave the map, but it will not affect a map file from which the map was loaded.

This is useful if you've read a map file into memory, made some mistakes, and want to start fresh.

Temporarily Leaving the Map

We can exit the map without destroying it (leaving it loaded in memory) with the command:

```
#map leave
```

This is useful when entering a maze or another room where the mapper will not work correctly.

Saving the Map to a File

Once a map has been created in memory it can be saved for later use with:

```
#map write <file>
```

Be very careful when saving your map files, this command will overwrite any file without warning. I have done this in the past (after a long session and very tired). I accidentally overwrote my command file rather than updating the map file.

To ensure this doesn't happen, I recommend creating an alias that always saves the file to the same place:

```
#alias mapsave {
    #showme "Map saved.";
    #map write mymap.map;
}
```

Two other tips when saving map files:

- You can make your map save alias automatically backup the previous map first using the #script command.
- You can setup automatic map saving using the #ticker command.

Loading the Map from a File

Maps that have been saved to a file can be loaded with the command:

```
#map read <file>
```

Just like the #map create command, this will not automatically insert you into the map.

Setting Our Position in the Map

To set our position in the map, use the command:

```
#map goto <vnum>
```

This will tell the mapper that we are at position <vnum> in the map. The goto command does not send any commands to the mud and therefore does not actually move us around. This is strictly used for telling the mapper where our position is.

If our map file is empty, the goto command will create the first room in the map. Once our map is populated, we can set our position to any room in the map using this command. Again, this doesn't actually move our character on the mud - it tells the mapper where our position in the map is.

This is mostly used when our actual position gets out of sync from where the mapper believes we are.

A quick note about vnums: Vnum's are unique integer's that function as an index for each room in our map file. When starting with an empty map (right after #map create, or if using #map read on an empty file), we have no rooms, so this will create the first.

Displaying the Map

If you've gone through this far, you're probably wondering where the map is. There are an infinite amount of ways to display the map, but we're only going to cover the most basic methods here.

The most basic way to display the map is through the command:

```
#map map
```

This command is used in other ways as well, but for our purposes right now, this will display the map. Of course, we don't want to have to type a command each time we want to display the map, we'd rather have always be displayed and updated.

Again, there are lots of ways to do this, and we're just going to cover the most basic here. TinTin provides a #split command, that allows us to split the screen. This is used for two primary purposes:

1. Separating the input field from the buffer
2. A very simple map display.

We're going to use it for both:

```
#split 16 1
```

You should now have a separate input field and 16 dashed lines at the top of the terminal window. In order to display the maps in the split, we need to use the command:

```
#map flag vtmap on
```

The #map flag vtmap on command takes advantage of this split to draw the map, only if the split is created and the flag is enabled. The map it draws is 16 rows high, and I don't believe there is a way to change this. Setting a larger height for the top split results in additional empty rows being displayed, but not a larger map display area.

After creating the splits, the buffer window is drawn differently than before. If you attempt to scroll up, you're going to see the buffer content you're looking for. Depending on your terminal you may need to use a different combination of keys to scroll correctly. On my OS X system in iTerm2, this is the function and up/down arrows.

Updating the Map

If you move around, you'll notice that the map is not updated - this is because the #map flag static flag is set and no other rooms exists (assuming you've got a brand new map with only 1 room).

We can disable this flag with the command:

```
#map flag static off
```

Now if you move around, you'll see that new rooms are created.

2.8.2 Editing the Map

Dynamic Mapping

As covered in fundamentals, we need to disable the static flag to allow rooms to be created as navigate around the mud:

```
#map flag static off
```

Once this is disabled, the mapper will automatically create rooms in the map as it interprets your movement.

NOTE: Dynamic mapping requires the #pathdir settings to be configured. The default are good when starting, but more can be done with these.

Manual Mapping

While the majority of mapping is typically done via dynamic mapping, there will be situations where you need to manually create or fix parts of the map.

To create an adjacent room, use the command:

```
#map dig <direction>
```

This will insert a room in the specified direction, but not move you to it.

To insert a room between you and an adjacent room, use the command:

```
#map insert <direction>
```

This will create a room between you and the room in the specified direction. This is used extensively for creating additional spaces to avoid overlap, and hidding entire map branches, covered in the following sections.

If you need to remove a room between your current room and the room on the other side of an adjacent room, you can use the command:

```
#uninsert <direction>
```

While the seems trivial, it's very useful because it automatically updates room vnums for both rooms and reconnects them for you automatically. Doing this by hand is very tedious.

There are also times when you need to move your position around the map, but not on the mud. This is done with the command:

```
#map move <direction>
```

This command is primarily used when the mapper gets out of sync with your actual location on the mud, especially while manually editing the map.

Undoing Room Creation

The mapper (out of the box) cannot tell when movement was prevented. It interprets your input and updates the map accordingly. One of the first things that happens in this situation, is that you'll attempt to move through a closed door or encounter some kind of situation that prevents the movement from occurring, but the mapper will create the room anyway.

There are several ways to recover from this situation, the first is with:

```
#map undo
```

This will unwind the erroneous room creation and move you to your previous location in the map. The second is to not unwind the room creation, but to move yourself back to your previous location in the map. This can be done with either of the commands:

```
#map goto <vnum>
#map move <direction>
```

If desired, you then delete the room that was created with:

```
#map delete <direction>
```

Overlapping Rooms

Most muds aren't laid out in a perfect grid - there's lots of overlapping. How you design your map is a personal preference, but one of the basic tools is to hide branches from the map.

Void rooms help to solve spacing issues by creating an additional "space" between two adjacent rooms.

The most basic way of using this, is with #map insert, and specifying the roomflag void:

```
#map insert <direction> void
```

This will move the room to <direction> over an existing space, creating the space needed to avoid overlapping rooms.

Hiding Map Branches

Sometimes, entire areas should be hidden from a view until they are entered. To do this, we use hidden void rooms. As previously covered, we can insert a void room with the command:

```
#map insert <direction> void
```

Note the vnum of the room that was created, because we will need to set the hide flag on it as well.

To do this, you create a "void" room (this is basically a place marker in the map) and set it's hidden flag to 1. When you navigate to this room you'll pass over it in whatever direction you were already going.

There are several ways to create the rooms, the most straight forward is:

> #map insert <direction> void #map goto <vnum> #map roomflag hide #map move <previous direction>

You should now see the entire branch hidden from your current position.

NOTE: It does not appear possible to set both the hide and void roomflags at the same time when using the #insert command. NOTE: THe #map return command does not appear to work to return from the void room to the previous location.

There are probably more efficient ways of doing this, but I don't know them.

2.8.3 Displaying the Map in a Separate Terminal

Rather than displaying in a split, we can get a much more enjoyable experience by displaying the map in a separate window.

The easiest way to do this is to have the MAP ENTER ROOM event write the map to a separate file, like this:

```
#event {MAP ENTER ROOM} {
    #map map 16x16 map.txt {a};
};
```

Note: The {a} flag appends to the file, rather than overwriting it, so that we can use tail of view it whenever it changes. This can generate very large files.

And then we can view the file in a separate window with tail -F

```
tail -n 16 -f map.txt
```

I've had mixed results with this approach - depending on the system and implimentation of tail, this may or may not work.

Rather than use this approach, we can get much better behavior by using the following bash script:

```
#!/usr/bin/env bash

MAP_FILE='map.txt'
MAP_SIZE='map_size.tin'
REFRESH_RATE=.25

while [ "true" ]; do
    echo \#var MAP_ROWS $(tput cols)\; > $MAP_SIZE
    echo \#var MAP_LINES $(tput lines)\; >> $MAP_SIZE
    clear
    cat $MAP_FILE
    sleep $REFRESH_RATE
done
```

This redisplays the file every .25 seconds, which on my system seems to be the fastest refresh rate that there is nearly no noticeable lag between my movement in TinTin and the map being updated. Additionally, it writes a small tintin file, which contains two variable definitions - the number of columns and number lines in the window displaying the map.

In our map event hook, we can then have it write the perfectly sized map, by simply reading this tintin file and using those values to update the size of the map that is displayed:

```
#event {MAP ENTER ROOM} {
    #read map_size.tin;
    #map map ${MAP_ROWS}x${MAP_LINES} map.txt;
};
```

2.8.4 Metadata

The mapper also has the capabilities for storing metadata about each room that can be used in an infinite amount of ways. The command #map info shows you some of these fields:

Room area: Room data: Room desc: Room name: Room note: Room symbol: Room terrain:

But map set shows them all (I believe):

> roomarea: roomcolor: roomdata: roomdesc: roomflags: 0 roomname: roomnote: roomsymbol: roomterrain: roomweight: 1.000

Any of these values can be set with the #map set command, for example:

> #map set roomarea mainland

These options are tailored to the MSDP protocol, which specifies values like TERRAIN that can be easily parsed and stored. I play on an LPMUD, which does not have terrain, so I don't use this value, but I do use the others.

2.9 Autologin

A typical feature of every config: game autologin. Usually it looks like this:

```
#event {SESSION_CONNECTED} {
  #showme {CONNECTED: %0};
  #regex {%0} {^sessionname$} {CONNECT name password} {0};
}
```

- `sessionname` is the name of your session you will use in the `#session` command
- `CONNECT name password` is the autologin command. Some MUDs understand only username and password, so you have to write: `username\npassword` instead.

This snippet activates on session connect. It checks the session name and sends the autologin command.

CHAPTER 3

Debugging

Debugging in TinTin is like hitting yourself in the face over, and over, and over. It hurts. And makes you feel dumb. These are some debugging tricks I've learned along the way that help make it a little less painful.

3.1 Verbose Configuration

TinTin's verbosity can be increased for debugging purposes with the command:

```
#config {VERBOSE} {ON};
```

However, this will not apply to command files loaded with the read command. These will need to be explicitly loaded with the command:

```
#line verbose #read <filename>
```

This will also increase the verbosity of the #DEBUG command, however that output always goes to the console and cannot be sent to a logfile.

3.2 Console Debugging

The included #DEBUG features provides some helpful data when needed. By default, debugging information is written the console and is enabled with:

```
#DEBUG {LIST} {ON};
```

To display all the debugging lists, use:

```
#DEBUG
```

To enable all the debugging lists, use:

```
#DEBUG {ALL} {ON};
```

The verbosity of the debugging information can be increased by enabling verbose config:

```
#CONFIG {VERBOSE} {ON};
```

Unfortunately, output generated by #CONFIG {VERBOSE} {ON} cannot be sent to log files.

3.3 Debug Logging

You can log debugging output to a separate file:

```
#log APPEND debug.log;
#debug {ALL} {LOG};
```

Note: #DEBUG's verbosity is increased by #CONFIG {VERBOSE} {ON} and will be sent to the log, but is also sent to the console. No way around this at present.

3.4 Debugging Command Files

If a command file is not behaving as expected - a useful debugging technique is to write the class out to a temporary file and review it. This very often exposes extremely common syntax errors - chunks of commands may be missing, identifying a position where to look for the source of the bug.

3.5 Extraneous Input

Sometimes you'll get the following message in your console when loading command files into the startup session:

```
#NO SESSION ACTIVE. USE: #session {name} {host} {port} TO START ONE.
```

Essentially, this identifies that some command file loaded some configuration that malfunctioned and attempted to write some data to the session, rather than run as a command. Unfortunately the information is not displayed or logged, so identifying what it was and what caused it can be tricky.

The following alias will capture all the extraneous input and write it to the console in bright colors:

```
#alias {%*} {
    #showme {<fab>MSG:INPUT       %0};
}
```

The downside to this technique is that it captures ALL input, that is not not a tintin command - all user defined commands, such as aliases and functions, will be captured and displayed rather than executed. This breaks most of my framework, so it's only useful on demand, and only after I've identified the immediate location of the malfunctioning code.

3.6 ANSI Bleeding

If colors are "bleeding", there is an included configuration setting that may resolve the issue:

```
#config {COLOUR PATCH} {ON};
```

3.7 References

- Adding Scope
- Scope Tricks
- Events Reading Closing Files

- Debug Framework Weird Behavior
- Verbose Classes

CHAPTER 4

Pitfalls

Issues that commonly trip up users.

4.1 Data Types

The data type documentation is significantly lacking. The table data type is completely undocumented. I highly recommend making sure you understand them thoroughly first, as the parser is extremely particular about how data is stored and referenced - and will fail silently if anything is out of place.

4.2 If Statement Terminations

When a statement is on a line all by itself, you need to (in most cases) terminate it with a semicolon (;). This is sometimes not required for block statements, such as #IF statements:

```
#if { condition } {
    commands
}
```

The previous code works fine - most of the time. However, if this code is in an alias and that alias is called from within an #if block somewhere else, the code will break ... for that occurrence only. The practical solution is to terminate all these code blocks explicitly:

```
#if { condition } {
    commands
};
```

4.3 Silent Failures

When issuing #read commands from an alias, all debugging output is silenced, regardless of:

```
#CONIG {VERBOSE} ON;
#DEBUG {ALL} ON;
```

You will not see error messages from these files. This present a significant problem for me, as I do all my loading through my module manager. You can force verbose output by prefixing the #read commands with #line verbose.

TODO: Test how #line log works with #line verbose.

4.4 Variable Collisions

There.is.no.scope. Variables using the same name in separate command files will overwrite each other. Beware.

4.5 Quoting Strings

When working with strings, quotes matter:

```
#showme "Test";          # yields "Test"
#showme Test;            # yields Test
```

These values are not equivalent. Typically, strings remain unquoted. Note: This has tripped me up several times when passing data between the shell and tintin using #script. Note: In some contexts, for example switch statements, strings MUST be quoted.

4.6 Switch Statements

Arguments in switch statements must be quoted, otherwise you'll get insane alias argument stack frame behavior. See "Quoted Strings" above.

4.7 Comments

If you don't terminate a #nop statement with a semicolon it will "absorb" the subsequent line:

```
#nop I'm just a comment
#script {
    ....
}
```

In this example, the script tag will be commented out and silently fail to execute.

4.8 Command Files

Command files must start with a #tintin command. If they don't, they'll fail to load. If you're loading the file through an alias it will silently fail to load. To make sure this never happens, I start every file with:

> #nop VSOF;

And regardless of how I modify my code, that line is never changed. Note: The error message displayed when this happens is "Invalid start of file." Hence VSOF (Valid Start of FIile).

CHAPTER 5

Cookbook

How to do different types of things.

5.1 Class Guards

When storing configuration and settings in command files, you can wrap them in “class guards” to ensure that the old code is destroyed each time the is reloaded:

```
#class {MYCLASS} {kill};
#class {MYCLASS} {open};
...
#class {MYCLASS} {close};
```

5.2 Clear Console

The following alias uses tput to clear the console from within tintin:

```
#alias {clear} {
    #system {tput clear};
};
```

5.3 Executing Code in All Sessions

Tintin includes the #ALL command for executing code in all sessions (#help all). Simply prefix #ALL to the command you want to execute in all sessions to use it:

```
#all #showme $session[name];
```

5.4 Parsing Tables by Reference

Through “passing by reference” it’s possible to parse a table without knowing it’s structure beforehand. First, we’ll create our example data structure:

```
#var V1.T1 {{T1K1}{T1V1}{T1K2}{T1V2}};
#var V1.T2 {{T2K1}{T2V1}{T2K2}{T2V2}};
```

Next, we create a variable that contains the name of the variable that contains our tables:: #var table_index V1;

Now we get a list of those tables. Note that we are using the "pass by reference" technique to operate on a dynamic variable name:

```
#var tables ${$table_index}[];
```

Now, we need to iterate over the tables. We've successfully abstracted ourselves away from operating on a specific table, so this will parse any set of tables stored in a nested variable:

```
#foreach {$tables} {table} {
    #var table {${$table_index}[$table]};
    #foreach {$table[]} {index} {
        #showme $index:$table[$index];
    }
}
```

5.5 Passing by Reference

Even though we don't have address or dereferencing operators in tintin, we can still emulate some very basic referential operations:

```
#var myvar     {mydata};
#var mypointer {myvar};

#showme $mypointer:${$mypointer};
> myvar:mydata

#alias myalias {
    #showme value of "dereferenced" alias argument: ${%1};
}
myalias myvar;
> value of "dereferenced" alias argument: mydata

#function myfunction {
    #return $%1
};

#showme valued of "dereferenced" function argument: @myfunction{myvar};
> value of "dereferenced" function argument: mydata
```

5.6 Read Verbose with Flag

Even with #CONFIG VERBOSE ON and #DEBUG ALL enabled, this output is still suppressed if #read is called from within an alias. We can get around this by prefixing the #READ command with #LINE VERBOSE. Unfortunately, if you have lots of files being read, then each time you want to enable this globally, you have to modify it throughout your entire configuration.

To make this a little more user-friendly, we can wrap the #read command in an alias that issues the command based on a global flag:

```
#var READ_VERBOSE 1;
#alias read {
    #if { "$READ_VERBOSE" == "0" } {#read %1};
    #else {#line verbose #read %1};
}
```

5.7 Reload Command File

When testing code, it's desirable to be able to run the same code over and over after making small changes. I add the following alias to my "test" command files when to make this very fast:

```
#alias reload {
    #kill {all};
    #showme <faa>Reloading ...;
    #read thisfile.tin;
}
```

NOTE: The reloading is taking place inside of an alias, so verbose output will be disabled.

5.8 Find the Length of a String

Find the length of a string:

```
#var thestring {something here}
#format resultvar {%L} {$thestring}
#echo {$resultvar}
```

References: - Finding the length of a string

5.9 Repeating Characters

To repeat a character a dynamic number of times:

```
#var cnt 30;
#format tmp %+${cnt}s;
#replace tmp { } {-};
```

References: - Repeat a character

5.10 Splitting a string by a set number of characters

Split a string into an array based on a set string length:

```
#format {test} {%w} {string}
```

Or use #regex to grab 80 or any number of characters at a time and split it.

References: - Split string into an array by width

5.11 Retrieving Table Indexes from a Variable

Retrieve a list of indexes from any reference point within a nested variable you must use the table index operator "[]":

```
#var MODULES {
    {module}  { {loaded} {1} {path} {modules/module/__init__.tin}  }
    {profile} { {loaded} {1} {path} {modules/profile/__init__.tin} }
    {session} { {loaded} {1} {path} {modules/session/__init__.tin} }
};

#showme $MODULES[];
{module}{profile}{session}
```

Note: You cannot retrieve this list of indexes and store them in a variable, they will be parsed as a table rather than a brace delimited list. The only place this works is within the context of a #foreach loop input.

5.12 Testing for the Existence of a Variable

To test for the existence of a variable:

```
#if {&MYVAR} { #showme $$MYVAR EXISTS         };
#else        { #showme $$MYVAR DOES NOT EXIST };
```

CHAPTER 6

Development

I'm trying to hack at the sphinx documentation a bit to add related content, integrated discussions via disqus, livereload support, and issue submission links for each page, so that an issue can be submit directly to the github issue tracker for documentation corrections.

I've got some notes below on adding these features.

6.1 Related Content

Add support for page tags and automatic interlinking. This way recipes in the cookbook page could be tagged by "feature" and automatically included in the feature pages.

- Finding Related Content with Sphinx
- Blogging with Sphinx

6.2 Disqus

Add disqus support for integrated discussions on specific documentation pages.

6.3 Live Reload

Add livereload support for automatic rebuilding of local wiki while editing.

- Adding LiveReload Support to Sphinx

6.4 Github Issues

Add links that allow users to directly submit github issues for documentation corrections.

CHAPTER 7

Contributing

I've invited all the TinTin++ user's on github that I know about to the organization, which should provide all of you (them) commit privileges. If I missed you, just sent a request to join the group here.

7.1 Documentation

The TinTin Unofficial Documentation is built using sphinx, which uses python. To build a local version of the documentation that you can edit and publish, you'll need to install the necessary python dependencies first. There are lots of ways to do this, personally I like to use virtualenv and virtualenvwrapper to setup an isolated python environment:

```
pip install virtualenv virtualenvwrapper
```

Then add the following to your shell initialization (.bashrc, .zshrc, etc) file:

```
if [[ -f '/usr/local/bin/virtualenvwrapper.sh' ]]; then
    export WORKON_HOME=$HOME/.virtualenvs
    export PROJECT_HOME=$HOME/Documents/Projects
    source /usr/local/bin/virtualenvwrapper.sh
fi
```

Then create the virtualenv and install the python requirements:

mkvirtualenv mudfiles pip install -r requirements.txt

And finally build your local copy of the documentation (from docs/):

```
make clean && make html && chrome build/html/index.html
```

Once you've make your modifications, if you've got commit privileges, just push your updates. Otherwise, submit a pull request.

7.2 RTD Theme

The docs use the sphinx rtd theme. I've modified the sphinx config.py to use this theme when building locally as well. To build locally, you'll need to install the them via pip (this is done automatically when you install the requirements).

I'm not sure yet, but I may need to fork the theme to setup the additional features on the Development page.

References:

- Read The Docs Theme Documentation

CHAPTER 8

Notes

- TODO: Figure out a way to add tags to sphinx pages, so “related” items are shown.
- TODO: Figure out a way to add a “fix me” button, so users can click a link on a sphinx page that redirects them directly to the issue tracker on github, so they can submit a correction.
- TODO: Figure out a way to integrate disqus so conversations can take place on specific pages.

8.1 Indices and tables

- genindex
- modindex
- search