LensC, a (hopefully) understandable guide. v2.x_0

Hi there. LensC can be daunting, especially for those with no prior programming experience. LensC can also be an incredibly powerful and useful tool for adding to the game. The purpose of this guide is to introduce and explain LensC in a way that even those of you who think "I'll never get this stuff" will get this stuff.

This guide was 99.999% written by Dredor. An absolutely amazing person, IC and OOC. Nothing but heart and grace. The other 0.001% is edits and updates by someone who couldn't be more different. A ghastly beast who hangs off the eaves of gothic churches known as Gimtor. Any new material added since Dredor's original text will be noted with a **[G] [/G]**.

LensC and OLC, a sidenote.

Before we get into writing LensC progs, let's touch on how we interact with them using OLC. It's fairly simple.

To add a program:

olc add prog

This will ask you for a Program Label. It's the name and short description of the program. It has no bearing on the actual program, just makes identifying or finding a program easier for us humans.

Some example Program Labels I've used:

Dredor [Immspell] Immspell quest anti-cheat code.

Dredor [Toy] Lift anything!

Dredor [Spell] Touch of Darkness. -30 saves + blindness.

Using simple and descriptive nametags like that makes finding any specific program of mine much, much easier than if I had left the field blank or put in something like "a program."

Once you choose a name, you will be sent into the editor. Here you enter in the text of your program. Use @ to end, just like any other time you're in the text editor.

Note: DO NOT LINEWRAP. No /$L, or whatever you use. Your program will not work.

To remove a program:

olc remove prog

To view programs in an area:

olc view prog

To edit a program:

olc edit prog <vnum>

As you can see, this is all simple stuff if you're at all familiar with building rooms, mobs, etc.

There is one special command you should know, however:

olc compile vnum

This **compiles** your program. If your program compiles, that means there are no apparent errors in it. (I say apparent because things can and do slip past the compiler.)

If your program does not compile, then you have done something wrong, and you must try and understand the compiler's sometimes cryptic error messages. That's a lesson in and of itself, and a topic for a future date.

[G] Focus on the first couple lines of errors the compiler gives you. For instance:

```
Preprocessor: including 384002 line 5.
Preprocessor: including 230033 line 6.
Error: Line 111, Token DOT        : Unexpected character
111:    int bar = 13.45;
Error: Line 112, Token COMMA      : Missing ')' in expression
112:    DEBUG(self, "bar: " + bar);
```

Everything after **111:** is errors resulting from the compiler breaking at line 111. So you have to figure out what it was at line 111 that failed. In this case, "Token DOT" refers to the decimal point in 13.45. LensC can't actually handle decimals. More on that later. Focus on the first error, fix that, then compile again and see what new horrors await you. **[/G]**

The Absolute Basics - What Every Program Must Contain

Below is an empty program. It will do absolutely nothing when run, but it won't generate any errors either.

program EmptyProgram (char)

main ()

{

}

And there we go, a perfectly valid LensC program. But... what does it mean? Let's take a closer look.

The first line is comprised of three parts. The word 'program', the program's name, and the program's caller type (the caller is what's running the program).

The word program is pretty self-explanatory. It has to be there, you put it there, and that's that.

The program's name is a single word that can be whatever you want. It's best to choose something descriptive. 'CheaterCheck' is a much more understandable name than 'DredorProg0546.' Just make sure it's a single word, 'Cheater Check' is not a valid name. 'Cheater_Check' works, though.

Finally, the caller type. This is either the word 'char' or the word 'obj', enclosed in parentheses. Which one do you use? Simple. Is your program going to be called by an object action, or a mob action?

Oaction? Use (obj)

Mob action? Use (char)

So, let's put that together. We've got some neat holy symbol object we made, and when players 'shatter' it we want their alignment to drop. This can be done easily via LensC. Let's write the first line for such a program. We know the first word must be 'program', that's a given. What shall we name it? Well, I'm not very creative so I'll go with 'PunishVandal.' I hope you come up with something better. Finally, I envision the holy symbol 'shattering' to be done via a spec_verb oaction we'd have on the symbol, so we would choose (obj) as the caller type.

program PunishVandal (obj)

And there you go. The first line, hopefully demystified. But what about that stuff below the first line of our EmptyProgram example above? Well, that stuff is the **main function**. What's that? It's where the body of your program goes. It's where the magic happens. When the game runs your program, it looks for the main function, then does the stuff inside those { } braces. You don't need to fully understand this just yet; we'll be returning to this later. But for now, what you need to know is that your program must have the following:

main ()

{

}

So, our vandal punisher will look like this:

program PunishVandal (obj)

main ()

{

}

And there we go. Of course, PunishVandal is just as empty as EmptyProgram was, but it's the very first building block in writing LensC programs.

If you'd like to try this out, do the following:

-Open a zone

-olc add prog

-Give your program a nametag.

-Enter in the text of the program (everything from the word 'program' down to the '}')

-@ out of the editor

-olc compile <program vnum>

You should get the following output:

Checking compiled types.

Parsing successful.

If so, congratulations, you've written a LensC program :)

Comments - What the Heck is Going On Here?!

Comments are text entered into the program that have absolutely no impact on the program.

Wait, if they do nothing, why have them?

To keep you, and others, sane. Think of them like notes you can scribble in the margins. Programs can get complex, and confusing. It's not uncommon to go back to one of your own programs and think "Huh? What? Why did I do ____ here?" Trying to understand someone else's program can be even more confusing.

To avoid this, you can leave a comment. Simply enclose your comment in /* and */.

Like this:

```
program ConfusingProg (char)

/*This program is empty. And confusing.*/

main ()

{

/*Dredor wuz here*/

}
```

Neither the 'This program is empty...' or 'Dredor wuz here' lines have any effect on our program. But any person looking at the program knows... well, that Dredor wuz here. Hopefully you leave more useful comments than that.

Comments don't have to be on their own line, either.

```
program ConfusingProg (char)  /*This program is empty. And confusing.*/

main ()

{

} /*Dredor wuz here*/
```

This works just as well.

Throughout the guide, I will be using comments to clarify on things.

Variables - An Introduction to Boxes

Often, we need to store information while our program is running. What information? All sorts of stuff. Could be the name of a character, the strength value of a mob, the room a player is standing in... the possibilities are endless. Where do we store this stuff? In variables, there's where. But, what the heck is a variable?

A variable is named storage space. Think of it like a box you can slap a label on. You stick your information in this box, and later on in the program, when you need that info again, it's sitting safely in that

box, waiting for you. There are a number of different types of variables, each one made to hold a specific kind of information. It's important to pick the right sort of box - you can't fit an elephant into a shoebox, and you probably should not try to.

Here's a brief overview of variable types.

int: Int stands for 'integer.' An int is a box that holds a non-decimal number. 72, 546, 78764. These can all fit in an integer box. 75.45, however, has a decimal place and is unwelcome. Trying to store it in an int variable will not yield expected results. **[G]** If you try to create an int in your program with a decimal, the compiler will break (as mentioned above in the section on the compiler). If you try to do some math that creates a decimal and assign that to the int, it'll just chop off any remainder.**[/G]**

bool: Bools are simple boxes. They hold one of two things: **TRUE** or **FALSE**. If your information either is or isn't (is a player flying or not, is a mob over level 10 or not, is this room named "A Quiet Office" or not, etc) than the bool is your box of choice. **[G]** Yes, the words must be all caps, TRUE or FALSE. LensC is case sensitive with **functions** and **constants** (constants are like variables that were created by Arawn and cannot be changed), meaning capitalization often matters.**[/G]**

string: Text! All sorts of text fits in the string box. "Yippee!" fits. "Dredor" fits. The name of the room Evarius is currently idling in fits. The long desc of any given mob fits. If it's text, you want a string variable to hold it in. Strings have a set of double quotes (" ") around them.

char: A character. This can be either a player or a mob. Dredor can fit in a char box. You can fit in a char box. The mind flayer? In the char box.

obj: An object. Whatever you're wearing on your head? It can fit in the obj variable. The desk in my office, the tree in New Lens heart, etc etc. All objects can fit in the obj box!

room: Are we getting the hang of this yet? If it's a room, it can fit in a room variable.

There's also exit, vehicle, and area variables. By now they should be self-explanatory.

Variables, Part 2 - Get Some Boxes, Fill Some Boxes

So, we've got boxes named variables, and different sorts of variables hold different sorts of things. Cool. How do we use them? Well, we must **declare** them. Declaring them is basically telling the game "Hey! Make a box of _____ type, name it _____." Like this:

int playerage;

Here we declare a variable of type **int** (so we know it's gonna hold a number), and we name it **playerage**. It seems reasonable to assume this variable is gonna hold a player's age. I didn't have to name it that though. We could very well declare it as *int derpderp;* and that's fine, but imagine reading someone else's program and all they use are nonsense variable names like that. Things would get confusing quick.

Please, pick descriptive and understandable variable names!

Also note the **;** at the end there. The **;** (called a **semi-colon**) basically says to the game "this is the end of this line." Most lensc lines will end with **;** though there are some exceptions as you'll see later.

Now let's put something in our box.

playerage = 52;

Notice we didn't need to put 'int playerage' again. We've already declared playerage, the program knows now that whenever we refer to playerage, it's an int. In this line we're saying "playerage equals 52." We're storing the number 52 in our little number box named playerage. If at some point later on in the program we look into playerage again, it's going to be holding the number 52.

We can condense those two lines into one, if we wanted.

int playerage;

playerage = 52;

is the same thing as

int playerage = 52;

In the first example we declare, then fill. In the second we do both at once. Both ways are correct; it's simply a matter of taste. I prefer the second, personally, if I know what it's going to be.

Let's declare some more variables.

*bool IsPlayerDead = FALSE; /*Create a variable of type bool, named IsPlayerDead, and fill it with FALSE.*/*

*string ImmName = "Dredor"; /*Create a variable of type string, fill it with the string "Dredor". Notice the "" around Dredor - these tell the program that "Dredor" is text, not the name of another variable or something.*/*

*room MyOffice; /*Create a variable of type room, name it MyOffice. We don't fill it with anything.*/*

Now, ponder this:

int BoxA = 12;
int BoxB = 8;
BoxA = BoxB;

What's going on here? Well, we make two ints. BoxA starts filled with the number 12. BoxB starts filled with the number 8. But that third line... "BoxA equals BoxB" What's going on there? Well, we're filling BoxA with BoxB. The 12 gets overwritten, and BoxB's 8 gets **copied** into BoxA. Copied, not moved. Remember that.

At the end of those three lines, **both** BoxA and BoxB hold the number 8. The 12 that BoxA started with is gone, BoxA holds only 8 now.

[G] This is a case where LensC is NOT case sensitive. *immname* and *ImmName* will be the same thing, so you can't get away with doing:

string ImmName = "Dredor";
string immname = "Gimtor";

If you were to check *ImmName*, it would say "Gimtor".**[/G]**

Functions - Arawn, Do the Work For Me!

Let's go back, for a moment, to the examples above where we were 'filling boxes.' Most of them are fairly simple. Int variables fit numbers, so when we fill them we pick numbers. 47, 5534, 10, etc. Strings, we pick strings. "Blah", "Dredor", "I wuz here". But, that room variable. How do we go about filling MyOffice with a room? We can't just say *MyOffice = Dredor's office!*, LensC won't know what on earth to make of *Dredor's Office!* It means something to us humans, but to the LensC compiler, it's gibberish. So how do we explain Dredor's office to LensC?

Functions. We use functions. What the heck is a function, you ask? A function is a bit of pre-written code that we can use. Arawn has provided a long list of functions that we can use to do -all sorts- of really neat stuff. At the moment the list of functions is at ~~http://www.lensmoor.org/imm_info/index.shtml though when the new site goes up that will move. Hopefully I'll remember to update this!~~ **[G]** https://lensmoor.org/castle_docs/lensc_f_new.shtml **[/G]**

Take a moment, go to that site, look through the functions. There are a **lot**, and they can be pretty scary looking if you're not familiar with them already. Don't worry, though, soon they will be your best friends. Back to our original problem at hand.

We want to fill the room variable MyOffice with Dredor's office. Dredor's office is room vnum 226000. We have a vnum, we need the room, and look at this, lurking in the list of functions:

function room get_room_by_vnum (int vnum)

That sounds like it might be useful...

This returns the room refered to by the given vnum. It can return an invalid room, so this must be checked.

Hrm, returns the room refered to by the given vnum. Why, that sounds like exactly what we want. So let's make some sense of this, shall we?

function room get_room_by_vnum (int vnum)

The first word, *function* is of no real use to us. We know it's a function, after all, we found it on the list of functions. The next word, *room*, is the type of variable this function will return. "Return" means the thing the function sends back. Not all functions send something back, but most do. We're looking to get a room

to fill MyOffice with, so this sounds pretty good so far. *get_room_by_vnum* - that there is the name of the function. When we use this function in our program, we'll use that name to refer to it. Finally, *(int vnum)*. This is called the function's **argument**. It's what the function needs from us to work. It needs an *int* variable that represents a vnum. That makes sense - if we don't give it the vnum of the room we want, it can't get anything done! Now, let's see how we'd use this.

```
program Get_Dredors_Office(obj)

/*I went with (obj), but since this example program will never be called by anything I could have gone with (char) instead.*/

main ()

{

  room MyRoom; /*Here we declare our variable.*/

  int OfficeVnum = 226000; /*Here we declare an integer, name it OfficeVnum, and fill it with 226000, the vnum of Dredor's office.*/

  MyRoom = get_room_by_vnum(OfficeVnum);

}
```

[G] Remember, the *obj* in Get_Dredors_Office(obj) just refers to the thing that will be running the program, which could be an object action or a mob (or player).**[/G]**

So, we've seen most of that before. This, however, is the new part:

```
MyRoom = get_room_by_vnum(OfficeVnum);
```

Let's turn this into people-speak. We fill the room variable MyRoom with the results of the function get_room_by_vnum, which we send 226000. The get_room_by_vnum peeks inside the OfficeVnum box and finds the number 226000. It does its magic and finds what room 226000 corresponds to. It then returns a variable that represents that room. We snag that return variable and stuff it inside our MyRoom box. MyRoom now contains room 226000. Not just the number 226000, but the -room-. Everything that comprises that room is in MyRoom. The description, the room title, the heal rate, the mana rate, the extra descriptions, the room flags, etc etc. There's a lot of things we can do with that room now, but we're going to go off on a very important tangent next.

Empty Boxes - Enemy of the People

In our last outing, we found the get_room_by_vnum function, sent it the vnum 226000, and got back a 'handle' to my office. If we'd sent it a different vnum, we'd have gotten back a different room. But hey,

what if we send it 226999? There is no room 226999! What would it return to us? What would happen? Let's take a look at the function's description.

This returns the room refered to by the given vnum. ***It can return an invalid room, so this must be checked.***

An invalid room? That sounds dire. It **is** dire! *MyOffice = get_room_by_vnum(226999);* would fill MyOffice with... well, with nothingness! With a room that doesn't exist. What if we then did things to this 'non-room'? What if we tried to change its room title, or move a player to it?

Explosions? Showers of sparks? The implosion of the mud?

Well, actually, usually your lensc program will stop. No message, no warning. It just rolls over and dies quietly. That's bad. Your program has crashed and you may not have the slightest idea why.

It's not just rooms that can be invalid, either. Objects, characters, areas... these are all potential landmines. How do we avoid these boxes full of nothingness? With our friend, the function. If you look on the function list page, you'll note several functions that begin with *is_valid_*. Let's take a look at one of them, *is_valid_room*.

function bool is_valid_room (room p)

The word *function* is as useless to as as ever, but the next word, *bool* is useful. As covered in our last lesson, we know that this is the type of variable the function will return to us. We know *bool* holds either TRUE or FALSE. Makes sense in this case - either the room we're testing is valid or invalid - there is no in-between 'sort of valid but not really' status, this is an all or nothing situation. We can see inside the () that the function takes a variable of type *room* - this is the room variable we shall be testing for validity. Let's see an example:

```
program test_this_room(char)

main ()

{

  room TestRoom = get_room_by_vnum(226999);

  /* Notice we declare a room variable, name it TestRoom, and immedaitely use the get_room_by_vnum function to fill it with the room at vnum 226999 */

  bool IsValid = is_valid_room(TestRoom);

/* We now declare a bool variable, name it IsValid, and we call the is_valid_room function, sending it our TestRoom variable. If TestRoom is valid, the function will return TRUE, otherwise it will return FALSE.*/

}
```

There we have it, a program that attempts to get a handle to a room, and stores whether or not that room is valid inside IsValid. Room 226999 doesn't exist, so in this case IsValid is sure to be FALSE. Of course, we didn't accomplish much here... Sure, we determined if the room is valid or not, but we never actually -do- anything with that information, do we? It'd be more useful to use this to raise some sort of alarm. "If IsValid is FALSE, then send an error message! Let the world know that this room is invalid and must not be used! Do something, man, do something!"

How exactly we make those kinds of decisions will be covered in our next exciting chapter! For now, just keep in mind that variables (excluding bool, int, and string) can be **invalid** and attempting to use them will cause your program to crash! Making use of the *is_valid* functions is vital, and something we'll go more in-depth into soon.

What Are We Going to Do Tonight, Brain? - Making Decisions

Often times you'll want to make decisions in your programs. You'll want to do things like "If this player is evil, have this mob attack!" or "If this room is invalid, issue a warning and quit the program." Up till now, our lessons have been fairly linear - the program starts, runs through each line, ends. No decisions to make. Let's give our programs some freedom, shall we?

Introducing, **if**!

```
program demonstrate_if(obj)

main ()
{
  int One = 1;
  int Two = 2;
  bool IsOneGreaterThanTwo = TRUE;


  if (One > Two)

    IsOneGreaterThanTwo = TRUE;

  else

  IsOneGreaterThanTwo = FALSE;

}
```

Whoa there, that's a lot to take in. Let's go line by line. The first important thing is we declare an int variable named One, and set its value to 1. We then create another int, named Two, and set its value to 2. Straightforward, no? We make a bool and name it IsOneGreaterThanTwo. We set it to TRUE. Still fairly straightforward, nothing new here.

The newcomer shows his face with *if (One > Two)*. Plain English? "If one is greater than two, do the next line." Taking a peek at the next line: *IsOneGreaterThanTwo = TRUE;* Okay, so "If one is greater than two, set IsOneGreaterThanTwo to TRUE." Hopefully this is fairly self explanatory, thanks to clear and helpful variable names!

Next line, another mysterious newcomer. Who is *else* and what are his plans? Well, *else* is an **optional** bit that can come after an if. Ifs do not require elses, but elses require ifs. Think of else as saying "Otherwise, do this next line." We can see that that next line would set IsOneGreaterThanTwo to FALSE.

So, all together: "If One is greater than Two, then set IsOneGreaterThanTwo to TRUE; otherwise, set it to FALSE." Of course, in the case of this example program, variable One will never be greater than variable Two. After all, 1 is never greater than 2, and those are the values we gave to our variables.

There is a lot more that can be said about ifs, a whole lot more, but next we're going to take a brief time out and slap together a sample program that will really help tie together much of what we've learned thus far.

Example Time!

```
program Our_Example(char)
 /* We shall have a mobact call this program, so we choose (char)
    Let us assume the mobaction looks something like mobact run 1000 % */

main ()
{
   char the_player;
   /*This will hold the player that we're dealing with. */

   the_player = get_char_room(self, argument);
   /*
      Here we want to fill the_player with the character who triggered this program. We're going to
      use get_char_room, which is basically "find a character in the same room as (variable 1),
      who has the name (variable 2). We want the player in the room with our mob, so we use self
      as the first variable. We used % in our mobact run statement, so argument should hold the
      player's name.
   */

   bool valid_player = is_valid_char(the_player);
   /* We need to make sure that the_player is valid! Maybe the player logged off just now, or left
      the room quickly. This is a must, to avoid any problems down the line. */

   if (valid_player == FALSE)
      end;
   /* Two new things here. == is NOT the same as =. == means 'is equal to', or 'do these two
      things match'. What we're asking here is 'is valid_char the same as FALSE?' If it is, we end;
      - we terminate the program immediately. A more robust program might find a way to output
      an error message, but we're keeping it simple.
      So - if the_player is not valid, we quit. */

   /* Now at this point we know that the_player is valid, as we would have ended the program just
```

```
    now if he wasn't. We can use him safely. */

  int player_age;
  /* This will hold the player's age! */

  player_age = ch_getval(the_player, CHAR_VAL_AGE);
  /* We want to get the player's age. A look-over of the function list will reveal ch_getval, a
     wonderful function able to provide all sorts of information on a character. The arguments we
     shall pass it are (person to get information from), (information we want). In this case we
     want info from the_player, and we want their age, so we send the_player and
     CHAR_VAL_AGE. CHAR_VAL_AGE comes from the function listing - it's not anything we
     defined or need to define. It is a constant the function requires. */

  if (player_age > 40)
    interpret(self, "say Man, you are oooooooold.");
  else
    interpret(self, "say You're not old at all. No sir, no sir.");

  /* Here we have the real meat of the program. Everything before this was just setting the
     stage. First - 'if player_age is greater than 40' - are they older than forty? If so, the next line
     happens. Interpret is another useful lensc function - it lets us force someone to do
     something. In this case we are forcing self, our mob, to tell the player they are old. Then
     comes else - if they are not over 40 we execute the next line, wherein we use interpret to
     inform the player that they are not old. */
}
```

And there you have it. This is a fully valid lensc program that you could paste in and have a mob run if you wanted to. It gets a handle to a player, makes sure that handle is valid, determines the player's age, determines if that age is over 40, and makes the mob react based on that information!

Now, there are a lot of ways this program could be improved, things that could be streamlined and whatnot, but if you can read through this and follow what is going on, you are well on your way to LensC proficiency!

Operators - Doing Stuff With Stuff

We've used **=** a bit now. **=** is called an **operator**, and there's more where that came from.

[G] This is actually where Dredor left off. I'll be adding what I can when I can. **[/G]**

Operators are symbols that do things to things. There are a lot, some of which you'll probably never use, others you will use all the time.

The first, and most common, is **=**. It's the assignment operator.

int One = 1;

Means "set the value 1 to the variable One". You can consider this to be a math operator. The other

common, basic math operators are *****, */*, **+**, **-**. These are muliply, divide, add, subtract. When you use these operators, the result will be a value of some sort, either an *int* or a *string*.

```
int result = One + Two * 3;
```

What do you think *result* will contain? Will it be 9? Or maybe 7? LensC follows an **order of operations** which dictates what instructions get done in what order. Your normal rules from math class apply here: *Please Excuse My Dear Aunt Sally*, or PEMDAS.

LensC first evaluates anything in ***p**arentheses*, then any ***e**xponents*, then ***m**ultiplication* and ***d**ivision*, then ***a**ddition* and ***s**ubtraction*. All other things being equal, it goes left to right. So in the above example, LensC will do "Two * 3" first, then add that with One. Your answer is 7.

Let's get more complicated!

```
result = One + Two * ((3 - One) / 2) + 4;
```

First the inner-most parentheses, so 3 - One is 2. Next the other parentheses take that 2 and divide it by 2, which is 1. Then we multiply that with the Two (because we multiply before adding/subtracting) and get 2. Then we go left to right and get One + 2 for 3, + 4 equals.. 7 again!

I don't expect you to memorize this; just know that order is important and parentheses are critical.

Next most common are logic operators. These are how LensC compares things, and they always return a *bool* of TRUE or FALSE. We already saw **==**, which means "are these two things equal to each other?" You can also ask, "are these two things NOT equal to each other?"

```
bool areEqual;
if (One != Two);
  areEqual = FALSE;
else
  areEqual = TRUE;
```

This asks, "If One does not equal Two, then areEqual is a damned lie." **==** and **!=** can also work on strings!

```
string wordOne = "one";
string wordTwo = "two";
string wordTmp = "";

if (wordOne != wordTwo)
  wordTmp = wordOne + wordTwo;
```

Now *wordTmp* will contain "onetwo".

Wait.. did I just add two *string*s together? .. Yup!