LPC For Dummies

Book Two

Michael Heron
(drakkos@discworld.atuin.net)

Beta Draft

Table of Contents

Mojo The Monkey Says...

If you find these texts useful and want to give less niche programming languages a try, come check out http://www.monkeys-at-keyboards.com for more free instructional material.

My apologies for the unfriendly legal boilerplate, but I have had people attempt to steal ownership of my material before.

Please direct any comments about this material to drakkos@discworld.atuin.net.

That's mojo at the top right. He's very clever. He has a B.A in Nanas!

The Mudlib Strikes Back

Introduction

Welcome to LPC for Dummies Two - the intermediate level material for Discworld Creators. Before progressing to this material, you should be sure that you understand the material presented in LPC For Dummies 1, **and** the Being A Better Creator material. For this text, the assumption will be that you have read both of these, and understand them all entirely. This is especially important for the coding side of things - seriously, if you don't understand ever single thing in LPC For Dummies 1, you shouldn't think about attempting this book. I assume a lot of you at this point.

Once we progress beyond the basics of LPC coding, it becomes possible to do some very fun things - however, the theory and practise gets considerably more complex when we start talking about commands and quests, so it's important for you to be willing to persevere. Before we progress to the Meat and Taters of this set of learning material, let's recap on what we're going to be doing as our worked example.

Betterville

We've set ourselves quite the challenge for our second learning village. We've set the theme of an area with a secret path that leads to a deserted, ruined tower. Within our tower, we have decided upon three quests:

- Sort the library
- Break through the secret area to the second level
- Find and unlock the secret passageway

In particular, we have set ourselves the task of making these quests, as far as is possible, dynamic. That, more than anything else, provides a substantial challenge to our development skills.

We have also decided on a range of NPCs to populate our area. First we have the Dozy Girls - wannabe princesses looking to cash in on a misunderstood local story. We have our shopkeeper, who is the lost lover of the beast and thus protected by his wrath.

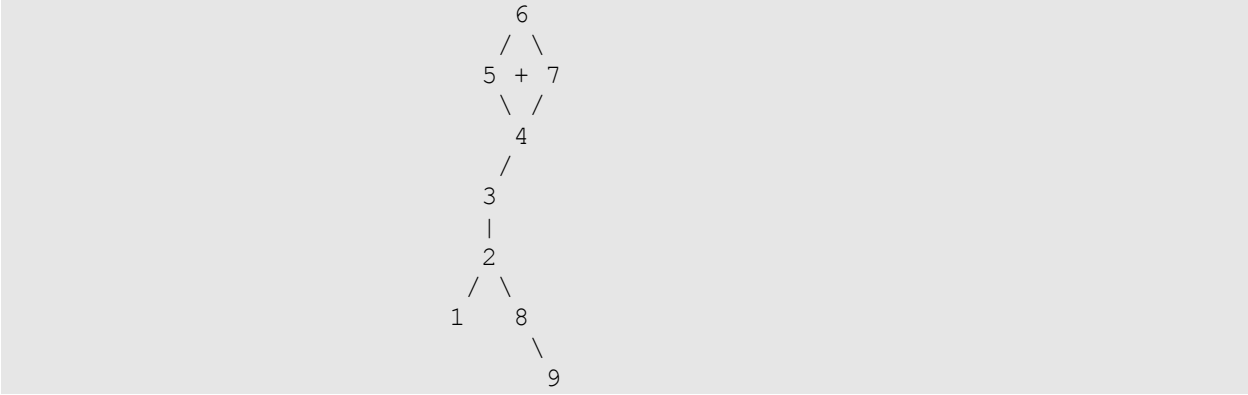
For our last NPC, we have the Beast - our star attraction. This is not a boss NPC but is instead a flavour monster that fleshes out the back-story. High interactivity is important here.

Finally, we have also decided on some features for the area. We have our library, which is the biggest draw to the village, and our local shop which is full of accessories for gold-digging teenage girls.

Along the way, we're going to look at building the coding infrastructure of this development. It has quite a complex make-up of features, and we want to be able to make sure it all works correctly and can be changed easily and efficiently. However, what we won't be doing is focusing on the descriptions very much - this is a coding manual, not a writing manual, and as with Learnville our focus is on making the development actually work, not with making sure it reads well. It also won't be 'feature complete' - we don't learn a lot by doing a dozen variations on the same theme, and so we will talk about how to do something once and leave it as exercise for the reader to fill in the blanks.

The Village Plan

We decided upon a layout quite early in Being A Better Creator - to recap:



Our first step is going to be to setup this framework. We're going to do it a little bit differently to how we did Learnville. This is LPC for Dummies Two after all - we do things Bigger and Better!

The biggest change we're going to make is in the architecture - we're not going to use the standard inherits for our rooms, we're going to create our own set of inherits. This gives us tremendous flexibility over shared functionality for rooms and NPCs. That will be our first goal with the village - putting the rooms together. You may think 'ha, I've already done that with Learnville', but not like this you haven't - trust me.

A Word Of Caution

The quests we write as part of this material are not supposed to be tutorials for how to write your own quests. We will cover certain key Discworld concepts in the process of building these, but you shouldn't think that any quest you write will be written the same way. The important thing in all of this material are the tools and techniques you use, rather than the product you end up building. This is an important point - you shouldn't think of any of this as a blueprint, it's just a process that is used to put together a complex area.

What you should be paying most attention to is the theoretical asides, such as when we talk about handlers, or inherits - essentially any of the 'science bits'. It is understanding when and where these programming constructs and design patterns should (and should not) be used that is by far the most important element of LPC for Dummies 2.

So please, don't just copy and paste the code that is provided - the code is the thing that is safest to ignore! **Why** the code was written in the way it was written though - ah, there be knowledge!

Conclusion

Fasten up tight kiddies, there's some pretty treacherous terrain ahead. By the end of this book, if you've understood everything on offer, you'll have the necessary toolkit to be able to code objects that will fascinate and delight young and old. You won't know all there is to know about LPC, but you'll know more than enough to be a Damn Fine Creator. Many creators over the years have been content to write descriptions and develop simple areas, and there is nothing wrong with that. However, when you want to do something that is genuinely cool, you need to know how the code for it is put together. That's what we're here for!

Inheritance

Introduction

When we built the infrastructure for Learnville, we made use of the standard inherits provided by the game for inside and outside rooms, as well as for NPCs. This is a solid strategy, but one that limits your options for adding in area-level functionality. In this chapter, we are going to build a series of custom inherits for our new and improved area. These will be specializations of the existing inherits (we're not going to have to have much code in them), but they'll make our lives easier as we go along.

Inheritance is one of the most powerful features of object orientation, and one of the reasons why the Discworld Mudlib is so flexible to work with as a creator. However, the cost of this is in conceptual complexity - it's not necessarily an easy thing to get your head around.

Inheritance

The principle behind inheritance is simple – it's derived from the biological principle of children inheriting the traits of their parents. In coding terms, it means that a child (an object which inherits) gains access to the functions and variables defined in the parent (the object from which it is inheriting). That's why functions like add_item work in the rooms we code. Some other creator wrote the add_item method, stored it in the code that defines a room, and as long as we inherit the properties of being a room in our own object, we too have access to that function. If you inherit /std/object, you will find the add_item function is no longer available.

At their basic level, inherits look something like this:

```
inherit "/std/outside";

void create() {
  do_setup++;
  ::create();
  do_setup--;

  // My Code In Here

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}
```

There is no requirement for an inherit to inherit anything itself, but since this is going to be our outside room inherit, we'll take advantage of having an object that already does all the work for us.

The create() method is common to all LPC objects – it's what the driver calls on the object when it loads it into memory. The code here is slightly abstract, so don't worry too much about what it means. In brief, what it's doing is making sure that setup() and reset() get called at the right time when an object is created. Without this, you get weird things like double add_items and such. Don't worry about it, it just has to be there.

Any code that we would normally put in the setup of an object can go between the two blocks of code, where the comment says 'my code in here'. For example, if you want every room in your development to have an add_item, you can do that:

```
inherit "/std/outside";

void create() {
  do_setup++;
  ::create();
  do_setup--;

  add_item ("betterville", "You better believe it!");

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}
```

Instantly, this makes life easier for us. We can have a common set of move zones, add_items, and even functions available to all the objects we create when we use this inherit this rather than /std/outside.

As a general note of good practise, store all your inherits together so people can easily find them – one of the side effects of it being possible to write them is that they can rapidly make it difficult to navigate through an object unless people know where they should be looking. For the sake of convention, all the inherits we talk about in this book will be stored in the /inherits/ sub-directory of our betterville folder. This one in particular will be stored as outside_room.c.

Hooking It All Up

Our next step then is to make it so this inherit is freely available in our project – in short, we setup our path.h file.

```
#define   INHERITS        BETTERVILLE + "inherits/"
#define   ROOMS           BETTERVILLE + "rooms/"
```

Now that we have this, let's put in place the architecture our our new village. We do this the same way as we did for Learnville, except we have a new and exciting inherit of which we can make use:

```
#include "path.h"

inherit INHERITS + "outside_room";

void setup() {
  set_short ("skeleton room");
  add_property ("determinate", "a ");
  set_long ("This is a skeleton room.\n");
  set_light (100);
}
```

Once again, we're going to do this for each of our outside rooms and add in the exits to link them up appropriately. You should remember how to do that from LPC for Dummies 1 - My First Area. If you don't, go back and read it. It's okay, I'll wait.

Okay, having done that, we have the basic skeleton of the village in place. Now, lo and behold, we can take advantage of our inherit by viewing our add_item in all the rooms!

```
> look betterville
Cannot find "betterville", no match.
```

Egads, what's this treachery? LPC, You go TOO FAR!

Replace Program

Well, there's a tiny catch here – the MUD has a useful little function called replace_program that it uses. What it actually does is far too technical to go into in detail, but essentially when you have an inherit chain this function looks over the chain for objects that don't **really** need to be part of it and skips over them. It decides this on the basis of whether or not an object has any functions defined in it. So it sees our inherit, thinks 'Nah, you don't need that' and skips over it. Our add_item then never gets added.

We don't want it to do this, so we can get around it in two ways:

- We can include the line set_not_replaceable (1) in the create function.

- Include a function as part of our inherit.

I favour the latter of these, for no real reason other than I can never remember the name of the function to call. So for now, let's add a function to our inherit:

```
inherit "/std/outside";

void create() {
  do_setup++;
  ::create();
  do_setup--;

  add_item ("betterville", "You better believe it!");

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}

int query_betterville() {
  return 1;
}
```

Now, when working with your own inherits, updating things gets a little more complicated. You first need to update the inherit, and then update each of the objects using it. You can also use the **dupdate** command, which updates each object in the inherit tree. Don't do that too often though, because you'll end up updating a lot more objects than you need to, and you never know when someone may be in the middle of working with a file you just carelessly updated mid-edit.

Success!

So, you update the inherit, and then update everything in your rooms directory. Magically, your add_item now appears in all its glory!

```
> look betterville
You better believe it!
```

Now we are well placed to put in whatever functionality we want in our inherit. We can add move zones (indeed, why not), light levels, room chats – anything we feel like. We can make an area-wide search function, or add commands to every room through the inherit. It's enough to make a person giddy with power!

```
inherit "/std/outside";

void create() {
  do_setup++;
  ::create();
  do_setup--;

  set_light (100);

  add_zone ("betterville");
  add_zone ("betterville outside");
  add_item ("betterville", "You better believe it!");

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}

int query_betterville() {
  return 1;
}
```

Having done this for our outside room, we should also make an inside room inherit. It's done exactly the same way except that we inherit a different object:

```
inherit "/std/room/basic_room";

void create() {
  do_setup++;
  ::create();
  do_setup--;

  set_light (100);

  add_zone ("betterville");
  add_zone ("betterville inside");
  add_item ("betterville", "You are inside and cannot see it from here");

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}

int query_betterville() {
  return 1;
}
```

Now, we can use this for each of our inside rooms in the same way we can for outside rooms.

However...

One of the things that makes LPC so powerful is that it's a language that supports the principles of multiple inheritance. This means that rather than having only one object that can be inherited from (as per Java and C#), LPC lets you define an object as inheriting from multiple different parents. That's extremely powerful, but also full of twisty little mazes and traps.

Look at our two inherits above – what happens if we want to share functionality between both of them? Like, for example, the query_betterville function. It's not much of a function, but if it were more substantial (like a piece of complex code attached to a search function), we wouldn't want to copy and paste the code for that. Instead, we make a shared inherit, and have our inside and outside rooms inherit **that** as well as the core Mudlib object.

This inherit isn't going to be an inside room or an outside room. Instead, it's going to just be an object we create from scratch – it inherits from nothing:

```
void create() {
  seteuid (geteuid());
}

int query_betterville() {
  return 1;
}
```

Notice our create method here – really, it isn't doing anything at all. We'll talk about euids later in the material, but you can think of that line of code as a kind of 'boilerplate'. That's the only line we need in create - there are no parent inherits it has to deal with. All we do here is define a function that we want to be available in both inside_room and outside_room. We'll save this shared inherit as betterville_room.c

We then make use of that inherit within inside_room and outside_room:

```
#include "path.h"

inherit "/std/outside";
inherit INHERITS + "betterville_room";
```

And:

```
#include "path.h"

inherit "/std/room/basic_room";
inherit INHERITS + "betterville_room";
```

This brings us back to our earlier problem with replace_program, so we change the query_betterville function with more specialized versions: query_inside_betterville and query_outside_betterville. Thus, an inside room has query_betterville (from betterville_room) and query_inside_betterville (from inside_room). Outside also has query_betterville, but query_outside_betterville instead.

Multiple Inheritance And Scope Resolution

Working with multiple inherits in one object causes problems with scope resolution. Specifically, imagine if you had a function with one name in one inherit, and a function with the same name but different functionality in a second inherit. When you call that function on your object, which one is supposed to be used?

There's a little symbol that we use to resolve this – you'll have seen it in the first inherit code we looked at, and also in LPC For Dummies 1:

```
do_setup++;
::create();
do_setup--;
```

The :: symbol is the 'scope resolution operator', and it tells LPC 'call this method on the parent object'. This is fine if there's only one parent, but when there is more than one, we need to refer to them specifically. In outside room:

```
do_setup++;
outside::create();
betterville_room::create();
do_setup--;
```

And in inside_room:

```
do_setup++;
basic_room::create();
betterville_room::create();
do_setup--;
```

The create methods get called in the order you give them, and the parents are differentiated by their unqualified filenames (their filenames without the directory prepended). If we only wanted to call one creator or the other, then we could do that too by omitting a call to the relevant parent.

Our Betterville Room Inherit

Now, the problem we have here is that since our betterville_room inherit doesn't inherit any of the room code, we can't use any of the normal room functions like add_item or such. Or rather, we can but we need to do it a little bit more awkwardly. If we try this, we will get a compile time error:

```
void create() {
  seteuid (geteuid());
  add_item ("stuff", "There is some stuff, and some things.");
}
```

However, we can do something like this instead:

```
void setup_common_items() {
  this_object()->add_item ("stuff", "There is some stuff, and some things.");
  this_object()->add_item ("things", "There are things, amongst the stuff.");
}
```

The this_object function allows us to treat our code as a unified whole - there is no add_item in betterville_room, but there is in the 'package' of betterville_room and betterville_inside_room, as an example. Using this_object() lets us call methods on the whole package, and not on the constituent bits.

In our inherits, we can make a call to setup_common_items as part of our code, and it will all work seamlessly. So, for outside_room:

```
#include "path.h"
inherit "/std/outside";
inherit INHERITS + "betterville_room";

void create() {
  do_setup++;
  outside::create();
  betterville_room::create();
  do_setup--;

  set_light (100);
  add_zone ("betterville");
  add_zone ("betterville outside");
  add_item ("betterville", "You better believe it!");

  setup_common_items();

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}

int query_outside_betterville() {
  return 1;
}
```

And for inside_room:

```
#include "path.h"
inherit "/std/room/basic_room";
inherit INHERITS + "betterville_room";

void create() {
  do_setup++;
  basic_room::create();
  betterville_room::create();
  do_setup--;

  set_light (100);
  add_zone ("betterville");
  add_zone ("betterville inside");
  add_item ("betterville", "You are inside and cannot see it from here");

  setup_common_items();

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}

int query_inside_betterville() {
  return 1;
}
```

Now we have an extremely flexible framework. If I want things that are common to outside rooms but not inside rooms, I put the code in outside_room. If they're for inside rooms only, they go in inside_room. If they should be shared between both, I can put the code in betterville_room.

Notice though that we'll have a problem if we want to create something that isn't an inside room or an outside room (like an item shop...). We'll come back to that later, but it's not a problem to add what specific inherits we need to make it all work seamlessly.

Visibility

Now, this system as it stands has a number of problems. Let's say for example that my inherit defines a variable. That variable gets inherited along with everything else, and so any object that makes use of mine can change the state of that variable without me being able to control it. Imagine a simple bank inherit:

```
#include "path.h"

inherit INHERITS + "inside_room";

int balance;

void create() {
  do_setup++;
  basic_room::create();
  betterville_room::create();
  do_setup--;

  if (!do_setup) {
    this_object()->setup();
    this_object()->reset();
  }
}

void adjust_balance (int bal) {
  balance += bal;
  if (balance < 0) {
    do_overdrawn(this_player());
  }
}

void do_overdrawn (object ohno) {
  ohno->do_death();
  tell_object (ohno, "Be more responsible with your money!");
}
```

Here, when someone goes overdrawn, they get killed as should be. However, there's nothing that requires people to go through this adjust_balance method. They can just directly manipulate the balance variable in the code they create:

```
inherit INHERITS + "crazy_bank_inherit";

void setup() {
  balance = -1000;
}
```

This is because our variable is **publicly accessible**, which is how all methods and variables are set as default. Public means that anything that can get access to the variable can manipulate it. I don't want people to be able to do this, because it circumvents the checking I wrote into my method. We can restrict this kind of thing through the use of **visibility modifiers**, which we briefly discussed in Working With Others. By setting a variable to be **private**, it means that it can only be accessed in the object in which it is defined:

```
private int balance;
```

Any attempt to manipulate it in a child object will give a compile time error saying that the variable has not been defined. They can define their **own** balance variable, but changing that won't change the state of the one in the inheritable. This forces people to go through our adjust_balance method. Any time we protect a variable in this way, we should expose a pair of **accessor methods** (a **setter** and a **getter**) that allow it to be manipulated with our consent:

```
int get_balance() {
  return balance;
}

void set_balance (int bal) {
  if (bal < 0) {
    do_overdrawn (this_player());
  }
  balance = bal;
}
```

In this way, we get the benefit of control over how a variable is manipulated, but we don't restrict the freedom of others to work with the internal state of our object.

The Impact Of Change

Remember how we talked about Impact of Change in Working With Others? This is why it's important - if we have a public variable in our inherit, we have to assume that somewhere someone is making use of that variable in their own objects, and if we change it we will break their code. That's bad voodoo, man. We limit variables to being private to keep our options open - if I want to change it from an int to a float, then I can't easily do it if it's set as being public. If it's private, then the only code that will break is in the inherit itself, and I can fix that directly.

For similar reasons, we may wish to restrict access to methods. We can do that too. Methods can be set to private - once they are, they are no longer possible to call from a child object, or from outside that object with call_other:

```
private void do_overdrawn (object ohno) {
  ohno->do_death();
  tell_object (ohno, "Be more responsible with your money!");
}
```

Sometimes though, we don't want to be this restrictive. Maybe we want people who are making use of our inherit directly to be able to access the function, but set it as inaccessible to outside objects. There is a 'middle' level of visibility that handles this - **protected**. It allows access for the inherit in which the method or variable is defined, **and** any objects that inherit that code. It does not allow access via call_other or the -> operator - it'll simply return 0 if access is attempted.

Conclusion

Putting in the skeleton of an area becomes much easier when we make use of bespoke inherits. We can bundle common functionality into one piece of code (thus making it much easier to maintain), and we can future-proof our development by making it easier for us to add wide-ranging systems at a later date without needing to change all the other rooms in the development. When you look at cities like Genua, Ankh-Morpork and Bes Pelargic, you'll see they were all written with this kind of system in mind.

It may seem like overkill to put an architecture like this in place for a small development like Betterville, but on those occasions where I have not done something similar for even small areas, I have regretted it. Take that advice in whatever way you wish!

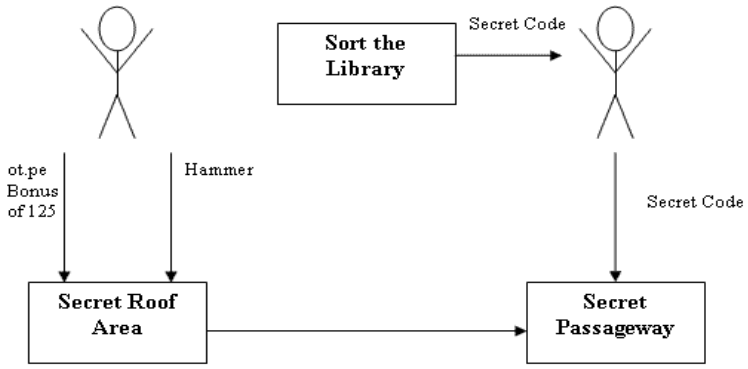
The Library Quest

Introduction

Let’s begin our discussion of the content with the library, our quest hub in this particular area. Our first quest, as I’m sure you recall only too well, was to sort the library shelves. We enter the room, and the floor is strewn with discarded books. There are bookshelves everywhere, each marked with particular category headings. Putting the books on the floor onto the right bookshelf is our quest.

It’s a solid quest – one that allows for us to put a dynamic framework around it so that it can’t be solved simply from a quest list. Because of that, it needs a little bit of thought to structure properly.

We also have an ‘output’ from solving this quest – it should give the secret code for the secret passageway quest that follows. Remember how it fits into our quest relationship diagram:



So, let’s get started!

Data Representation

Of all the decisions that a coder makes, how they choose to represent the data is the most important. It’s what changes an impossible task into a trivial task, or - importantly - a trivial task into an impossible task. If you choose good data structures for how you store and manipulate all of the data associated with a system, your job becomes exponentially easier than if you choose bad data structures.

When we talk about data structures, we mean the combination and interrelationship of data-types within an LPC object. An array of mappings is a data structure. A mapping of mappings is a data structure. Ten ints and a string is a data structure.

Our choice in this respect should be influenced by several factors:

1. How easy is the data structure for others to read and understand?
2. How efficient is the data structure in terms of memory and CPU cost?
3. How simple is it to query the various parts of the data structure?
4. How simple is it to manipulate the various parts of the data structure?

The requirements for each of these factors will vary from situation to situation. Core Mudlib code must be highly efficient and simple to query and manipulate. It doesn't have to be especially readable since the people likely to be working with it are usually more technically experienced. On the other hand, code written for these tutorials must emphasise readability, and readability is one of the key predictors of how maintainable code will be. For normal, every day domain code where you can't assume a certain level of coding knowledge, readability is a hugely important feature.

Our decision as to how to represent our data will emerge out of a consideration of the data itself – what do we need to store, for how long, and in what ways do we need to manipulate the data?

Let's consider our library quest.

- We need to store the titles of some books.
    1. We need to be able to get a random list of these
- We need to store some category headers.
    1. We need to be able to get the list of these
- We need to store how a player has sorted books
    1. We need to know which category a player has associated with each book
    2. We need to let players move books from category to category
    3. We need to let players reset their entire categorization.
    4. We need to let players add books to categories
    5. We need to let players remove books from categories

It’s apparent here that the first two sets of data require little manipulation, and the last set of data requires considerable manipulation. If we change the parameters of our quest, we also change the nature of the data representation. For example, say we let people add their own books (a bad idea, but let’s say we did). In addition to the requirements above, we would also need to allow players to add titles to the list of books, and remove titles from the list of books. We’d also need to be able to save the data – note that we don’t need to do that at the moment.

Changing the parameters of the quest will have an impact on how simple the data is to manipulate. If you’ve chosen a good data structure for your purposes, it will be possible to add new functionality as time goes by with minimal fuss. If you’ve chosen a bad data structure, then new functionality becomes very difficult to implement. That's what we mean by 'maintainability'.

Let’s look at two ways of representing this data. First, a straw-man to show the impact of bad data representation:

```
string book1;
string book2;
string book3;
string book4;
string category1;
string category2;
string category3;
string category4;
mapping books_and_categories;

void create() {
  seteuid (geteuid());
}

void setup_quest() {
  book1 = "Some stuff about you.";
  book2 = "Some stuff about me.";
  book3 = "Some stuff about her.";
  book4 = "Some stuff about him.";
  category1 = "romance";
  category2 = "action";
  category3 = "erotica";
  category4 = "comedy";

  books_and_categories[book1] = category1;
  books_and_categories[book2] = category3;
  books_and_categories[book3] = category2;
  books_and_categories[book4] = category4;
}
```

This kind of data representation does not lend itself well to modification or expansion. What happens if you want to add in twenty new books? What happens if you want books to belong to more than one category? What happens if you want people to be able to sort the list of books so they can browse through it easier? None of these things are simple to do with this bad data structure. A slightly better one:

```
string *books;
string *categories;
mapping books_and_categories;

void setup_quest() {
  books = ({"Some stuff about you.", "Some stuff about me.",
    "Some stuff about her.", "Some stuff about him.");
  categories = ({"romance", "action", "erotica", "comedy"});

  books_and_categories[books[0]] = categories[0]
  books_and_categories[books[1]] = categories[2];
  books_and_categories[books[2]] = categories[1];
  books_and_categories[books[3]] = categories[3];
}
```

It's still not great – we have to hand roll the connection between each book and each category. How about this:

```
mapping books_and_categories;

void setup_quest() {
  books_and_categories["romance"] = ({"Some stuff about you"});
  books_and_categories["comedy"] = ({"Some stuff about me"});
  books_and_categories["erotica"] = ({"Some stuff about her"});
  books_and_categories["action"] = ({"Some stuff about him"});
}
```

Here, we’re going a slightly different way – if we ever want a list of the categories, we need to pull it out of the data structure like so:

```
string *query_categories() {
  return keys (books_and_categories);
}
```

We’ve traded off a little efficiency in favour of a more elegant representation. Now, if we want to add a pile of new books, it’s a trivial task:

```
void setup_quest() {
  books_and_categories["romance"] = ({"Some stuff about you", "Gnoddy",
    "Things that go bump in the day"});
  books_and_categories["comedy"] = ({"Some stuff about me",
    "Where's My Cow?"});
  books_and_categories["erotica"] = ({"Some stuff about her",
    "Things that go bump in the night"});
  books_and_categories["action"] = ({"Some stuff about him",
    "The Nevewending Story"});  }
```

There are better ways still to do this – our structure is efficient, but it's not especially expandable. What happens if we want to make the books a bit more interesting? If for example we wanted to add the name of the author, or a blurb on the back?

That's not easy to do with our current representation – we need something different.

In A Class Of Your Own

LPC makes available a special kind of 'user-defined' data type called a class. For those of you with any outside experience of object orientation, please don't make any assumptions about the word 'class' - it's nothing like a class in a 'real' object oriented language. With a class, you create a single variable that has multiple different compartments in it. For example, we could do this:

```
class book {
  string title;
  string author;
  string blurb;
  string *categories;
}
```

With this definition we've created a brand-new data type for use in our object – we can designate things as being of type 'class book':

```
void setup_quest() {
  class book newBook;
}
```

We create new variables for a class in a different way from most variables – we use the **new** keyword:

```
newBook = new (class book);
```

Once we have the new 'instance' of this class, we can set its individual constituent parts using the -> operator:

```
void setup_quest() {
  class book newBook;

  newBook = new (class book);
  newBook->title = "Some stuff about you";
  newBook->author = "You";
  newBook->blurb = "An exciting tale of excitement and romance!";
  newBook->categories = ({"romance"});
}
```

You can also combine the steps of creating a new class and setting its elements like so:

```
class book newBook;

newBook = new (class book,
  title: "Some stuff about you",
  author: "You",
  blurb: "An exciting tale of excitement and romance!",
  categories: ({"romance"))
);
```

A single variable of a class is useful, but when combined in an array format they become especially powerful. What we get is something akin to a simple database. Imagine the following data representation:

```
class book {
  string title;
  string author;
  string blurb;
  string *categories;
}

class book *allBooks = ({ });

void add_book (string book_title, string book_author, string book_blurb,
  string *book_categories) {

  class book newBook;
  newBook = new (class book,
    title: book_title,
    author: book_author,
    blurb: book_blurb,
    categories: book_categories);

  allBooks += ({ newBook });
}

void setup_quest() {
  add_book ("Some stuff about you", "you",
    "A exciting tale of excitement and romance!", ({"romance"}));
}
```

Now, when we want to add a new book, we just call add_book – as many books as we like. However, if we want to get a list of categories, we have traded off the simplicity of arrays for something that needs a more bespoke solution:

```
string *query_categories() {
  string *cats = ({ });
  for (int i = 0 ; i < sizeof (allBooks); i++) {
    foreach (string c in allBooks[i]->categories) {
      if (member_array (c, cats) == -1) {
        cats += ({ c });
      }
    }
  }

  return cats;
}
```

A structure like this requires some management functions for ease of manipulation. For example, what if I want to get the blurb that belongs to a particular book? I need a way of finding that book element in the array, so I need a function to do that:

```
int find_book (string title) {
  for (int i = 0; i < sizeof (allBooks); i++) {
    if (allBooks[i]->title == title) {
      return i;
    }
  }
  return -1;
}
```

Then, if I want to query the blurb of a book:

```
string query_book_blurb (string title) {
  int i = find_book (title);

  if (i == -1) {
    return 0;
  }
  return allBooks[i]->blurb;
}
```

There’s some more upfront complexity here, but the trade-off is that it’s much easier to add in new functionality as time goes by. If we want to add in a new set of information to go with each book, we just add it and it’s there and supported. Things get a bit trickier when we load and save classes, but that's a resolvable issue.

The Library State

So, that sets us up with something that stores each of the books. How do we represent how the player has currently sorted the library? Here, we need to answer a new question – do we store this information about the player, or do we store it about the room?

If we store it about the player, it means that multiple players can be working in the same library without their actions impacting on the state of the other players. If we store it about the room, then all people working within the library have a shared library state.

Really, there isn’t a right answer here – but since it’s a little bit weird if everyone is working within their own 'meta' library, let’s just decide we’re going to store the state about the room – so when one player sorts a book on a shelf, it changes the state of the room for everyone else in it. That means we need some more variables in our room to store which books are where. We can store that pretty simply as a mapping – book title X is on bookshelf Y:

```
mapping sorting;

int assign_book_to_shelf (string book, string category) {
  int i = find_book (book);
  string *cats;

  if (i == -1) {
    return -1;
  }

  cats = query_categories();

  if (member_array (category, cats) == -1) {
    return -2;
  }

  sorting[book] = category;
}
```

Believe it or not, that's the bulk of the engine of the quest done. Everything else proceeds simply from this point because we have a solid data representation we can rely upon.

Dynamic Quest Design

Ideally, a dynamic quest will involve some kind of random setup of the books. For example, let's say that our book titles do not give the clue to what the category is, only the blurb does. With this comparatively minor modification to the concept, we can then randomly assign each book a category (and thus a blurb) when we add it. Well, that's no problem at all:

```
void add_book(string book_title, string book_author) {
  class book newBook;
  string my_blurb;
  string *cats = ({"romance", "action", "childrens"});
  string my_cat;
  string *valid_blurbs;

  mapping blurbs = ([
    "romance" :
      ({
       "A romantic tale of two estranged lovers",
       "A heartbreaking tale of forbidden love!",
       "A story of two monkeys, with only each other to rely on!",
       }),
    "action" :
       ({
       "A thrilling adventure complete with a thousand elephants!",
       "An exciting story full of carriages-chases and brutal sword-fights!",
       "A story of bravery and heroism in the trenches of Koom Valley!",
       }),
    "childrens":
       ({
```

```
        "A heart-warming tale of a fuzzy family cat!",
        "A story about a lost cow, for children!",
        "A educational story about Whiskerton Meowington and his last "
          "trip to the vet!",
       })
  ]);

  my_cat = element_of (cats);
  valid_blurbs = blurbs[my_cat];
  my_blurb = element_of (valid_blurbs);
  newBook = new (class book,
    title: book_title,
    author: book_author,
    blurb: my_blurb,
    categories: ({my_cat}));

  allBooks += ({ newBook });
}
```

Now we have a quest architecture in place – we give in a list of titles and authors of books, and it sets up the details randomly from that. People will need to read the blurb to find out what the book is about and then sort it into the appropriate category. There's no way to short-cut this quest, you need to actually put in the work yourself. Sure, someone may quest-list all of the blurbs and the categories to which they belong, but it's pretty obvious from the blurb anyway. There is simply no value in quest-listing this.

We can add in as many books as we like here, but we want to set some kind of sensible limit for our players – say, ten books to give a reasonable amount of work to do before the quest is awarded.

The Rest Of The Quest

This is the core of the quest, but we also need to provide the bits around the edges – we need to be able to give lists of information on request. For example, the list of all books that have not yet been sorted:

```
string *query_unsorted_books() {
  string *unsorted = ({ });
  for (int i = 0; i < sizeof (allBooks); i++) {
    if (!sorting[allBooks[i]->title]) {
      unsorted += ({ allBooks[i]->title });
    }
  }
  return unsorted;
}
```

A way of telling to which category a book has been sorted:

```
string query_bookshelf_of_book (string title) {
  return sorting[title];
}
```

And a way to reset the sorting if it's all gone horribly wrong:

```
string reset_sorting() {
  sorting = ([ ]);
}
```

There may be more we need, but these will do for now. We can't anticipate everything, after all.

This may not look like any quest with which you are familiar, and that's because it's not – something like this sits behind every quest in the game, but that's for our eyes only. For the players, we need to give an interface to our code – a way for them to call functions in a sanitized manner. For example, we may give the player a command in a room:

```
sort book_title into category <category>
```

Upon typing this command, it hooks into the functions we've written – specifically, it will do the 'assign_book_to_shelf' function, providing the book title and category they give. We need to then give meaningful output to the player indicating what has happened. That's for another chapter though.

Conclusion

This chapter has introduced a new, powerful data-type to you – the class. I am a big fan of classes, they make almost everything easier to do. However they are syntactically distinct from the other data types with which you will be familiar, and so you should make sure you are comfortable with how they work before you start playing about with them.

Data representation is a hugely important topic – get your representation wrong, and the rest of your code is doomed to be a hopelessly unreadable, unmaintainable mess. Get it right, and good code will flow like honey. That's just the way of it.

Adding Commands

Introduction

We've built the bulk of the technical architecture for our first quest, but what we need to do now is build the user interface to that architecture. The phrase ‘user interface' usually conjures up images of windows based applications, but all it means is ‘what sits between my code and my users'. In the case of a MUD, the user interface may be an item, it may be a room, it may be a command, or it may be something else.

What we need to decide then is how we are going to allow our user to interact with the functions we have written. As with most things, this will be influenced by the context of the code – there is no definitively right situation.

Deciding On A User Interface

There are certain elements of building a user interface that are universal rules to which we must adhere. In fact, it's not a mandate – we have all sorts of things in the game that violate these. However, if you break them knowingly then someone (that is, me) will cut you.

A good user interface has the following traits:

- Consistency
- Predictability
- Allows for undoing mistakes
- Provides meaningful feedback to users

There are other traits consistent to all user interfaces, but these are the ones that we need to view as iron-clad rules for Discworld development.

A good user interface is consistent. That means that if similar or identical functionality is available in a different part of the game, our functionality should mirror the way it's accessed. For example, if we have a mail-room in every city in the game, then it's violating user interface design if one is accessed through commands and another requires you to talk to an NPC. We violate this rule all the time, but those violations should be considered bugs – it's not okay to go against consistency. We should always adopt the best interaction choice and be consistent with how it is applied.

Predictability relates to how the cues that we provide in the game should be interpreted. Imagine if we had a parcel that when unwrapped yielded not a gift but instead a package of spiders that ate your face. That may be funny, but it's not justifiable unless there is some kind of predictability built into the parcel. Perhaps if you look at it close enough you get a clue as to what is within, or if you leave it long enough you can see it moving, or hear legs within. If you provide a course of action to the user, the result must be knowable otherwise you are punishing people for exploring your code.

As a second example of this, imagine a situation in which we have a game full of red doors and green doors. The red doors do some damage to you and yield a reward, and the green doors permit access with no downsides. This is part of the user interface since it is from this that the user builds their vocabulary of experience. If you then suddenly half way through the game reverse these colours, then the effect is you tell your player ‘Anything you think you have learned may be discarded at any time'. The user then has no way of making meaningful decisions regarding their choice of interacting with the game.

It is important that for everyday usage there is a mechanism for undoing changes. We have a ‘high consequence' model for certain things in the game – rearranging is easy to do and hard to undo, and you can't easily undo a bad choice in learning skills. This is fine, but something to do sparingly. For day to day interaction with the game, there should be an easy way to undo anything that a player has done. For our quest, we make it easy for people to reassign books from category to category, as well as start over.

Finally, your user interface should provide meaningful feedback to a player. Imagine if whenever we try to shelve a book incorrectly we get the following message:

```
> Try again.
```

How do I choose to make a meaningful decision here? Is the problem with my syntax? Is it with the category I'm using? Am I not using an existing book? Is the book already shelved?

There is no way I can get from this error message to a useful change in my behaviour. Thus, when you provide feedback to the user it should indicate in a meaningful way whether there is success or failure. It should also speak in language the player is likely to understand. ‘Object reference is inconsistent with expected parameters' may make sense to a developer, but a player would much prefer something like ‘that particular item isn't what you need here'.

Quests on Discworld don't have a cast-iron interaction convention, but the majority of them are accessed through commands defined in rooms or items or NPCs. Because that's a common interaction context, that's the choice we too will make. In order to do this, we need to talk about a new Discworld construct, the add_command.

Adding Commands

You've already done this in a simple way in LPC for Dummies one, where we added commands to add_items. Discworld offers a powerful system for making available commands to your players, and it's called add_command. Alas, its power ensures that it's one of the hardest things to use properly, so we must spend a considerable amount of time talking about the theory behind the function.

At its basic level what it lets you do is provide a command that exists only in a certain context and as long as a certain condition is true. Usually that context is while the player is either:

- Inside the object (rooms)
- Holding an object (items and NPCs)
- In the same environment as the object (items and NPCs)

You don't need to worry about keeping track of this, it's all done for you by the Discworld parser. All you need to know is how to add the commands. It's the parser that is responsible for taking what a user types into the game and breaking it into different parts. You don't need to worry yourself overly about this, just treat it as a bit of magic for now – just imagine it as that a player types in a string such as ‘stab drakkos with knife' and the parser breaks that up into the command (stab), the objects involved (drakkos and knife) and the rest of the text which is just for aiding in making our commands a little more ‘natural' to type in.

The exact way in which the parser does this is decided by the **pattern** we give the command we add. We can decide specifically what kind of arguments that an add_command should accept and work with.

Traditionally, an add_command is located in the init method of the object with which you are working. Let's look at one simple example of this before we get into the specifics.

```
void init() {
  ::init();
  add_command ("test", "<string>");
}

int do_test(object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  printf ("Test: %s\n", args[0]);
  return 1;
}
```

In init, we add the command – it's called ‘test' and the pattern is any string of text – it matches anything that follows the word ‘test'. When we use the command, the MUD looks for a function called do_COMMAND, where COMMAND is what we named the command. We can change this default behaviour, but we won't talk about that just yet.

When we enter a command in this way, the function do_test gets called, with all the parameters in the parameter list above (they get handled for you). In this function, we simply print out the text that the user typed while in the room:

```
> test This is a test!  Test: This is a test!
```

Each of the parameters to the function holds a different piece of information. We'll talk about indirect and direct objects later – for now the only one we are interested in is *args*. This array holds each of the different parts of the command that followed the name. Since we only have a string, that's all it shows. However, consider if we had the following as a pattern:

```
void init() {
  ::init();
  add_command ("test", "<string> with the <string>");
}
```

The pattern acts like a filter and a ‘fill in the blanks' system – the user must type out this pattern exactly, but they can put whatever they like in the dynamic bits – the bits marked with <string>. So the following would all be valid commands:

```
test drakkos with the cakes
test cakes with the tea
test tea with the toasting forks
test a whole load of people with the standard examination
```

However, none of the following would be valid:

```
test drakkos
test with the toasting forks
test drakkos with the
```

When we try to enter an invalid combination of command and text, we get the standard error message:

```
See "syntax test" for the input patterns.
```

When the do_test function gets called with a valid command string, our code above only picks up the first string that was part of the match:

```
> test here with the thing  Test: here
```

Args thus contains an array of each specific 'blank' that the user gave to our pattern. I'm sure you can see why that is useful!

Now, let's extend our simple example a little by incorporating meaningful user feedback. We do this through the use of a method called add_succeeded_mess:

```
int do_test(object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  add_succeeded_mess ("$N $V the command.\n", ({ }));
  return 1;
}
```

The $N and $V are examples of tokens that get processed by the MUD and turned into meaningful output depending on who is seeing the message. You will see:

```
You test the command.
```

Other people would see:

```
Drakkos tests the command.
```

The $N gets replaced with the short of the person performing the command, and the $V gets replaced with the verb used (in this case, ‘test'). Pluralisation is done automatically depending on who is doing the observing.

The final thing we need to discuss about this simple example of an add_command is the return value. This is meaningful – a return of 1 means that the command succeeded and that a succeeded message (added by us) should be displayed. If it returns 0, it means the command failed and that a failed message should be displayed, like so:

```
void init() {
  ::init();
  add_command ("test", "<string>");
}

int do_test(object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  if (args[0] == "pass") {
    add_succeeded_mess ("$N $V the command.\n", ({ }));
    return 1;
  }
  else {
    add_failed_mess ("You failed the test.\n", ({ }));
    return 0;
  }
}
```

A failed message goes only to the player, and should be used when the actual functionality of a command has failed to begin - for example, if they were using inappropriate objects, or they used an inconsistent pattern. When you return 0 from an add_command function, the MUD keeps checking for another object that may be able to handle the player's input. If you return 1, it stops checking for another object to match the command. In this way, multiple objects can define commands with the same name, and only those appropriate to a situation will be triggered.

We'll talk about the array provided to each of the add_x_mess methods a little later in this chapter.

More On Patterns

The real power of add_command comes from the fact we are not restricted to matching strings - we can also make it match other things. A list of matching codes and the things they are used for:

Code	Matches
<indirect>	An object
<direct>	The object in which the function associated with a command is defined.
<number>	A number (which must be expressed in digits)
<word>	A single word
<fraction>	A fraction, such as 3/4, 5/6, etc
<preposition>	Such as ‘to', ‘from' etc

We can also specialize these patterns, depending on what they actually are. We can do so by separating following specializations with a : . Usually you do this for objects rather than the other types, but the help file for add_command will give you some things you can do with strings and numbers.

For indirect and direct, you will often want to specify what kind of object you are are trying to match. The default is any object at all, but you can override that by providing a second part to the match:

Code	Matches
object	Any object
living	A living object
distant-living	Uses the function find_living to match an object.
any-living	Matches living objects in the room first, and then distant objects later. Also allows for the user of ‘someone’, ‘everyone’ and ‘creators’ as valid matches
player	Only matches a player
wiz-present	Wiz present allows for the complex creator level matching. See ‘help wiz-present’ for details.

We can specialize our patterns with a colon, so:

```
<indirect:living>
```

Or

```
<direct:object>
```

We can also further specialize these by specifying where a match should be made from:

Code	Matches

me	The inventory of the living object performing the command
here	The inventory of the environment of the living object
here-me	Check the environment first, and then the inventory
me-here	The inventory of the object, and then the environment.
direct-obs	The inventory of the direct object

So, if I was in the strange situation of wanting to match any living object in my inventory, I could use the following:

```
<indirect:living:me>
```

Or if I wanted to match players in my environment only:

```
<indirect:player:here>
```

The syntax of the command gets built automatically from our pattern (so when players do ‘syntax command', they get the proper information). However, we can make them more readable by providing a label that gets used instead of the pattern:

```
<indirect:living:me'pets'>
```

The text in apostrophes is what will be displayed to the user instead of the pattern itself.

Optional Parameters and Set Choices

We also have the option in a pattern of providing optional parameters (ones that allow the command to flow properly), and choices from a restricted set. Optional parameters are indicated by square brackets, and set choices are indicated by braces:

```
add_command ("get", "<string> from [the] thing");
```

The ‘the' is not required in order to make the pattern match, but it won't cause a problem if it's present. The optional parameters get ignored – they don't get sent into the parameters of the function.

If we want to provide a set of choices, we can do that too:

```
add_command ("paint", "wagon {red|blue|green}");
```

This will match any of the following:

```
paint wagon red
paint wagon blue
paint wagon green
```

Nothing else will match. The exact option the player chose will be included in the args array that gets passed as the fourth parameter to the function.

Direct and Indirect Objects

The different between these is not necessarily obvious. The direct object is the one in which the function for handling the command resides – it will almost always, unless you are doing something clever or weird, be this_object(). For example, if you have a command defined in a room, then the room will be the direct object. If you have a command defined in a weapon, then the weapon will be the direct object. My advice is, don't worry about it until such time as you encounter a situation where an indirect object isn't working the way in which you need.

You could use it productively in the meantime as a resolution for which object to which you are referring. If you're wearing a pair of boots:

```
add_command ("kick", "<indirect:living> with <direct:object>");
```

This would work almost the same as just using a string ("<indirect:living> with boots") except that it allows for more complex pattern matching and gives you the object of ensuring that the boots are being worn and such.

Indirect objects are everything else – anything that's used by a command but isn't the object in which the command is defined.

Passed Parameters

Now that we've spoken a bit about how add command works, let's talk about the parameters that get passed to the function. There are five of these, and they each contain different information.

The first parameter is an array of indirect objects – all the objects that matched the pattern that we set. So if we have a pattern of <indirect:object:me> and someone uses ‘sword' for that, it gives us a list of all the objects in the player's inventory that matches the word ‘sword'.

If we have more than one check for indirect objects in our pattern, we get an array of arrays instead, each array containing a list of objects that matches the pattern. For example:

```
add_command ("cut", "<indirect:living > with <indirect:object:me>");
```

Our player has this command, and types ‘cut drakkos with knife'. Our first parameter then would contain an array of arrays, the first containing all the living objects that match the name ‘drakkos', and the second containing all the objects in the player's inventory matching the name ‘knife'. These get passed in the order in which the objects were matched.

The second parameter is the string that was matched for any direct:object pattern. You can safely ignore this for now.

The third parameter is the list of string matches for indirect objects. Ignore that too for the moment.

The fourth parameter contains all the arguments that the user provided to the command – they're the blanks that the user filled in. They get populated in order, so if the pattern is:

```
add_command ("test", "<string> with <string> and <string>");
```

And the command is:

```
test bing with bong and bang
```

The args array will be:

```
({"bing", "bong", "bang"});
```

Finally, the last parameter is the pattern that matched the user input. If a command has many different patterns (which is common in guild commands and such), this is the one that met the requirements of the user input.

Tannah's Pattern Matcher

Adding commands and working out what the parameters are is horrendously complicated. Luckily, Our Creator|Tannah}), back in the dawn of the world, wrote a wonderful tool for helping you get it clear in your head – Tannah's Pattern Matcher, which may be found in /d/learning/items/matcher.

The matcher allows you to add commands to it and then execute them – as a result, it gives you what the parameters are for the command as added. Let's look at a simple example of using it:

```
> add command bing with pattern "<indirect:any-living>"
You add the command "bing" with the pattern "<indirect:any-living>" to
Tannah's Pattern Matcher.

> bing citizen
Indirect objects: ({ /* sizeof() == 1 */
  /d/forn/genua/chars/citizen#5116424 ("citizen")
})
Direct match: 0
Indirect match: "citizen"
Args: ({ /* sizeof() == 1 */
  "citizen"
})

Pattern: "<indirect:any-living>"
```

I honestly can't emphasise enough how useful this can be to see what the impact of various kinds of patterns are when using add commands. My advice is to clone yourself one of these and try it out with every combination you can imagine until you are 100% sure of what the patterns are and how they work. It'll make the rest of your life as a creator so much easier.

When you read the matcher, it will show you what commands you have added:

```
You read Tannah's pattern matcher:

The pattern matcher is currently set to test the following commands and
patterns:

[0] "bing", "<indirect:any-living>"

See 'syntax add' and 'syntax remove' to modify the list.
```

And you can remove commands with ‘remove command <number>'.

Make this tool your friend, you won't regret it!

Conclusion

Add command is the tool that lets you add worlds of functionality to your objects. As befitting a complex, powerful function it's not the easiest thing in the world to learn how to use. However, once you have mastered it you will find it a breeze to add complex, interesting functionality to everything you create.

Next we'll look at how we use add_command to build the user interface for our library quest, so make sure you understand what we've spoken about in this chapter. Read it over a few times if the next chapter doesn't make sense - the assumption will be that commands, functions, the parameters they use, and the patterns are all clear to you.

The Library Room

Introduction

Now that we've spoken in some depth about add_command and how it works, we can start to build the front-end of our library quest. For this, we need a room (the library) which will contain the usual Room Related Things as well as the code we developed to handle the manipulation of the books and categories. It's our job now to craft a compelling environment in which players can easily participate in the quest.

The Room

The room we're going to develop is going to inherit from the inside room inheritable we created earlier. We need to do a little retrofitting and expansion to make our room work properly, but our starting point will look like this:

```
#include "path.h"
inherit INHERITS + "inside_room";

class book {
  string title;
  string author;
  string blurb;
  string *categories;
}
class book *allBooks = ({ });
mapping sorting;

void reset_sorting();
void setup_quest();

void setup() {
  set_short ("main library");
  add_property ("determinate", "the ");
  set_long ("This is the main part of the library.  It's a mess, with "
    "discarded books thrown everywhere!\n");
  set_light (100);
  add_exit ("west", ROOMS + "betterville_09", "door");
  setup_quest();
  reset_sorting();
}

void add_book(string book_title, string book_author) {
  class book newBook;
  string my_blurb;
  string *cats = ({"romance", "action", "childrens"});
  string my_cat;
```

```
  string *valid_blurbs;
  mapping blurbs = ([
    "romance" :
    ({
      "A romantic tale of two estranged lovers",
      "A heartbreaking tale of forbidden love!",
      "A story of two monkeys, with only each other to rely on!",
    }),
    "action" :
    ({
      "A thrilling adventure complete with a thousand elephants!",
      "An exciting story full of carriages-chases and brutal sword-"
        "fights!",
      "A story of bravery and heroism in the trenches of Koom Valley!",
    }),
    "childrens":
    ({
      "A heart-warming tale of a fuzzy family cat!",
      "A story about a lost cow, for children!",
      "A educational story about Whiskerton Meowington and his last "
        "trip to the vet!",
    })
  ]);

  my_cat = element_of (cats);
  valid_blurbs = blurbs[my_cat];
  my_blurb = element_of (valid_blurbs);
  newBook = new (class book,
    title: book_title,
    author: book_author,
    blurb: my_blurb,
    categories: ({my_cat}));

  allBooks += ({ newBook });
}

string *query_categories() {
  string *cats = ({ });

  for (int i = 0 ; i < sizeof (allBooks); i++) {
    foreach (string c in allBooks[i]->categories) {
      if (member_array (c, cats) == -1) {
        cats += ({ c });
      }
    }
  }

  return cats;
}

void setup_quest() {
  add_book ("Some stuff about you", "you");
}

int find_book (string title) {
  for (int i = 0; i < sizeof (allBooks); i++) {
    if (allBooks[i]->title == title) {
      return i;
    }
```

```
  }

  return -1;
}

int assign_book_to_shelf (string book, string category) {
  int i = find_book (book);
  string *cats;

  if (i == -1) {
    return -1;
  }
  cats = query_categories();

  if (member_array (category, cats) == -1) {
    return -2;
  }

  sorting[book] = category;
}

string query_book_blurb (string title) {
  int i = find_book (title);

  if (i == -1) {
    return 0;
  }
  return allBooks[i]->blurb;
}

string *query_unsorted_books() {
  string *unsorted = ({ });
  for (int i = 0; i < sizeof (allBooks); i++) {
    if (!sorting[allBooks[i]->title]) {
      unsorted += ({ allBooks[i]->title });
    }
  }
  return unsorted;
}

string query_bookshelf_of_book (string title) {
  return sorting[title];
}

void reset_sorting() {
  sorting = ([ ]);
}
```

Phew, that’s quite a lot, but it sets us up nicely for the code to follow. To begin with, let’s set the scene for our players. We need to ensure they have a way of getting all information they need through interacting with the room. First of all, they need to know what the books are that need sorted. Let’s tie that into an add_item. Because it’s going to be dynamic as to what the add_item should say, we need to use a function pointer (we'll talk about these at the end of this text). The skeleton for that would be as follows:

```
string books_to_sort();

void setup() {
  set_short ("main library");
  add_property ("determinate", "the ");
  set_long ("This is the main part of the library.  It's a mess, with "
    "discarded books thrown everywhere!\n");
  set_light (100);

  add_item ("book", (: books_to_sort() :));
  add_exit ("west", ROOMS + "betterville_09", "door");
  reset_sorting();
}

string books_to_sort() {
  return "blah";
}
```

Now, we need to build our books_to_sort() function. It should have the following behaviour:

- If there are books that have not been shelved, we should list those books
- If there are no book left to shelf, it should indicate that with a special message.

Lucky ducks that we are, we already wrote a method to do the bulk of this work for us – query_unsorted_books. We just need to tie that into this method:

```
string books_to_sort() {
  string *unsorted = query_unsorted_books();

  if (!sizeof (unsorted)) {
    return "All the books have been neatly sorted away.";
  }
  else {
    return "The following books are strewn around the library: " +
      query_multiple_short (unsorted);
  }
}
```

So, that’s step one – giving the player the cue as to what remains to be done. The next step is to provide a way to find out what’s been done so far – specifically, what books are currently on which bookshelves. There are many ways we could do this:

- Allowing players to look at each bookshelf directly
- Looking at bookshelves gives the state of all books in all categories

- Looking at books individually tells which bookshelf they are on.

I favour a single add_item providing all information, because I feel it gives maximum usability with minimal frustration. So let's add a bookshelf item:

```
add_item (({"shelf", "bookshelf"}), (: books_on_shelves() :));
```

We also need a function for handling the text for when the player looks at the shelves. Our sorting mapping actually contains this information already, so we just need to process it and print it out 'real nice like':

```
string books_on_shelves() {
  string ret = "";

  if (!sorting || !sizeof (sorting)) {
    return "No books have been sorted onto the shelves.";
  }
  else {
    foreach (string book, string category in sorting) {
      ret += "The book \"" + book
      + "\" has been put on the shelf for the category \"" + category
      + "\"\n";
    }
    return ret;
  }
}
```

The final piece of information we need to provide is what categories are actually available. We could build that into the bookshelves, or provide a cue to look at another add_item. I favour the latter as it's not information that is constantly updated and we risk overloading a player by bundling all the information into the same place. So we change our bookshelf add_item a little:

```
string books_on_shelves() {
  string ret = "The bookshelves are arranged according to categories.  ";

  if (!sorting || !sizeof (sorting)) {
    ret += "No books have been sorted onto the shelves.";
  }
  else {
    foreach (string book, string category in sorting) {
      ret += "The book \"" + book
      + "\" has been put on the shelf for the category \"" + category
      + "\"\n";
    }
  }
  return ret;
}
```

And then we add in an item that gives us the categories:

```
string book_categories() {
  return "The following categories are labelled on the bookshelves: " +
    query_multiple_short (query_categories());
}
```

Now all that we need is a mechanism for allowing players to shift books from the main pile to a particular category, and for viewing the blurb on a book. A pair of add_commands would do nicely here.

Guess The Syntax Quests

I am as guilty of this as anyone, but we must be mindful when coding an add_command to make it so that the syntax is obvious. Guess the syntax quests are frustrating for everyone, and usually revolve around a creator having picked an unusual word for a command (often without realizing it wasn't intuitive), and then providing no hints later as to what the right command actually is.

We need to avoid these, but it's not easy. One good way is to provide multiple command syntaxes that point to the same function, or just make sure you pick clear words from the start. Hints as to the right words to use should also be liberally sprinkled around your quest. For example, if we were to use the command 'sort', our book item might suggest that:

```
string books_to_sort() {
  string *unsorted = query_unsorted_books();

  if (!sizeof (unsorted)) {
    return "All the books have been neatly sorted away.";
  }
  else {
    return "The following books are strewn around the library: " +
      query_multiple_short (unsorted) + ".  They are just begging to "
      "be given a good sorting.";
  }
}
```

If you design a properly dynamic quest, you might even consider making the instructions available in a help-file, or on a web-page. The logical process you go through to solve the quest should be the task, not finding the actual commands you need to use.

Viewing The Blurb

Our books are not items. They could be, but they're not – players can't hold them, and they certainly can't open them up and read them. However, 'read' is a command that will automatically be attempted if people want more information from the book, and so we'll provide a 'read blurb on book' command. We should also give the hint to players that there are blurbs to read:

```
string books_to_sort() {
  string *unsorted = query_unsorted_books();

  if (!sizeof (unsorted)) {
    return "All the books have been neatly sorted away.";
  }
  else {
    return "The following books are strewn around the library: " +
      pretty_array_output (unsorted) + ".  They are just begging to "
      "be given a good sorting.  Each has a blurb on the back that "
      "may help in discovering to what category they belong.";
  }
}
```

So, first we add the command – it goes into the init method:

```
void init() {
  ::init();
  add_command ("read", "blurb on <string'book'>");
}
```

And then we add the code that handles the functionality. It's not complex, we already have methods to do everything we need to do:

```
int do_read (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string book = args[0];
  string blurb;

  blurb = query_book_blurb (book);

  if (!blurb) {
    add_failed_mess ("That book does not seem to exist.\n");
    return 0;
  }

  tell_object (this_player(), "The blurb on the book reads: " +
    blurb + "\n");
  add_succeeded_mess ("$N pick$s up a book and read$s the back.\n",
    ({ }));
  return 1;
}
```

With that, the blurb has been dealt with. Except, not entirely – it doesn't take into account language skills or such. We'll come back to that later.

The Sort Command

Now that we can read the blurb on the back of a book, let's give ourselves a sort command. As before, we put it in the init method of our room. It'll need to take in two pieces of information – the book we want to sort, and the category into which we want to sort it:

```
void init() {
  ::init();
  add_command ("sort", "<string'book'> into [category] <string'category'>");

  add_command ("read", "blurb on <string'book'>");
}

int do_sort (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string book = args[0];
  string category = args[1];

  return 1;
}
```

So, when our player wants to take a book and sort it into a category, they type:

```
  sort book_name into category category_name
```

Or just:

```
sort book_name into category_name
```

The syntax message should hopefully be self-explanatory:

No need to guess the syntax, it's all there wrapped up in the command. That's good practise – a bad syntax is bad for everyone.

So, what do we need to do in our command? Well, we need to provide a way of giving useful feedback to the player – for example, if they are trying to sort a book that doesn't exist, or into a category that isn't there, we should tell them that:

```
int do_sort (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string book = args[0];
  string category = args[1];

  if (find_book (book) == -1) {
    add_failed_mess ("That book does not seem to exist.\n", ({ }));
    return 0;
  }

  if (member_array (category, query_categories()) == -1) {
    add_failed_mess ("That category does not seem to exist.\n", ({ }));
    return 0;
  }
  return 1;
}
```

Finally, if it turns out that they are trying to sort a valid book into a valid category, then let it be so:

```
int do_sort (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string book = args[0];
  string category = args[1];

  if (find_book (book) == -1) {
    add_failed_mess ("That book does not seem to exist.\n", ({ }));
    return 0;
  }

  if (member_array (category, query_categories()) == -1) {
    add_failed_mess ("That category does not seem to exist.\n", ({ }));
    return 0;
  }

  assign_book_to_shelf (book, category);

  add_succeeded_mess ("$N sort$s the book \"" + book +
     "\" into category \"" + category + "\"\n", ({ }));
  return 1;
}
```

Finally, we need to add in a function that checks to see if the quest has been completed – if our current library state matches the desired end-state (which is that all books are in the right category). That's easy to do, since we have a pretty adaptable data structure in place. We need to check the following:

- If there are any unsorted books, then the quest is not complete.
- If there are any books in the mapping that don't exist in the database (which shouldn't happen, but it's nice to make sure), we bail out.
- If a book is shelved in an incorrect category (as in, one that it doesn't have in its category array), we bail out.
- If none of these things are true, the quest is complete.

So, we translate that into a method:

```
int check_complete() {
  int i;
  if (sizeof (query_unsorted_books())) {
    return 0;
  }
  foreach (string book, string category in sorting) {
    i = find_book (book);
    if (i == -1) {
      return 0;
    }
    if (member_array (category, allBooks[i]->category) == -1) {
      return 0;
    }
  }

  return 1;
}
```

Finally, we hook this function into our sort command:

```
int do_sort (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string book = args[0];
  string category = args[1];

  if (check_complete()) {
    add_failed_mess ("The library is immaculately organised.  Don't go "
      "spoiling it now.\n", ({ }));
    return 0;
  }

  if (find_book (book) == -1) {
    add_failed_mess ("That book does not seem to exist.\n", ({ }));
    return 0;
  }

  if (member_array (category, query_categories()) == -1) {
    add_failed_mess ("That category does not seem to exist.\n", ({ }));
    return 0;
  }

  assign_book_to_shelf (book, category);

  if (check_complete()) {
    tell_object (this_player(), "If this were an active quest, you "
      "would have just gotten a wodge of XP!\n");
  }

  add_succeeded_mess ("$N sort$s the book \"" + book
    + "\" into category \""  + category + "\"\n", ({ }));
  return 1;
}
```

And voila – one quest, coded and ready to go.

Except, not quite...

Having gotten the core of the quest done, we need to do some fine-detail work to make it clear and simple to use. For one thing, we have a fairly awkward user interface problem:

Oh no! The reason for this is that we have capitalization in our book dataset, and so not only the letters must match, but the case of the letters must match too. We can resolve this by liberal use of lower_case and cap_words throughout our code. As a convention, it is always best to store string data in lower case (unless casing is important) and capitalize it as it is shown to the user. So when we add a book, we could add it like so:

```
void add_book(string book_title, string book_author) {
  ...
  book_title = lower_case (book_title);
  ...
}
```

And then when the user enters the book and category to search, we do the same:

```
string book = lower_case (args[0]);
string category = lower_case (args[1]);
```

Now we have no need to concern ourselves about the capitalization of user input. It's all in the same format. However, it will look quite bad if we always output in lower case, so we can make sure that the 'user-facing' representation gets a little bit of polish before it gets to the player. For example, in books_on_shelves():

```
else {
  foreach (string book, string category in sorting) {
    ret += "The book \"" + cap_words (book) +
      "\" has been put on the shelf for the category \""
      + cap_words (category) + "\"\n";
  }
}
```

And in our do_sort:

```
add_succeeded_mess ("$N sort$s the book \"" + cap_words (book)
  + "\" into category \""  + cap_words (category) + "\"\n", ({ }));
```

There are some areas of our code where that's slightly tricky to do, such as when we're working with query_multiple_short and arrays. We can easily write a method that handles that for us though:

```
string pretty_array_output (string *arr) {
  for (int i = 0; i < sizeof (arr); i++) {
    arr[i] = "\"" + cap_words (arr[i]) + "\"";
  }

  return query_multiple_short (arr);
}
```

And then use it in place of each of the calls we have to query_multiple_short, like so:

```
string book_categories() {
  return "The following categories are labelled on the bookshelves: "
  + pretty_array_output (query_categories());
}
```

And:

```
string books_to_sort() {
  string *unsorted = query_unsorted_books();

  if (!sizeof (unsorted)) {
    return "All the books have been neatly sorted away.";
  }
  else {
    return "The following books are strewn around the library: "
      + pretty_array_output (unsorted) + ".  They are just begging to "
      "be given a good sorting.";
  }
}
```

These kind of helper functions can greatly simplify the task of building a compelling user experience in your rooms, and also greatly increase the consistency of your interface. If we only occasionally use quotation marks, then we make our quest harder to do because people can't trust the visual cues. We don't want that – either we do it everywhere, or we do it nowhere. In between is worse than not doing it at all.

Now, back to our blurb. As it stands, it works – but it doesn't work properly because we have a fairly complex language system on Discworld. What happens if someone who has no knowledge of Morporkian tries to read our blurb? They shouldn't be able to do so. Luckily, our message system has a nice way of allowing you to turn any arbitrary bit of text into language specific text, like so:

Not only that, but we can also distinguish between read and spoken languages by adding a *read:* flag:

So therein lies our solution:

```
int do_read (object *indirect_obs, string dir_match, string indir_match,
  ...

  tell_object (this_player(), "The blurb on the book reads: "
    + "$L$[read:morporkian]" + blurb + "$L$\n");

  ...
}
```

And then Bob is the brother of your mother or your father!

Conclusion

That's a quest we just did. It's easily the most involved thing we've done in any of this material, but hopefully you'll have seen it wasn't too bad, Indeed, it's actually a good deal more complicated than many quests – it has random elements that allow it to be entirely dynamic – plus, the more books we add, the more variation can come into the quest.

This isn't a quest room as it would necessarily appear in the game – instead, it's a starting point that you can play about with. There are many, many ways to make this quest better than it is. Really, the limit is only your imagination.

This has been quite a complex chapter, and we haven't yet touched on the output of this quest – the secret code that feeds into the quests to follow. We'll discuss that later. For now, let's bask in the warm glow of a hard job well done!

Quest Handlers

Introduction

The code that we have in place now describes the actual quest, but we still have to actually make the quest available. This requires us to do two things:

- Have the quest added to the quest handler
- Hook the code we have written into the player library

In this chapter, we'll look at the process of setting up our code to interface with the two handlers dedicated to managing quests.

The Handlers

There are two handlers that exist to manage the complexity of quests. The first of these is the **quest handler** itself, which holds information about each of the quests such as what they are called, the hints associated with them, and the level. The second handler is the **library**, and the library holds the records of a specific player with regards to the quests they have done and any quest info that has been associated with them.

You don't need to worry particularly about the quest handler – it's your domain lord that will have to add the quest and make it available to players. There's a web-interface you can explore at http://discworld.atuin.net/lpc/secure/creator/quests.c to see what information the quest handler contains. Specifically, check out the Betterville Library Quest – that's the placeholder quest entry for the code we've done.

The library is what we'll spend most of our time manipulating. Before we can do that, we need to include library.h to make available the library handler to our code.

First, let's look at the one place in our code that currently has to handle the awarding of the quest:

```
if (check_complete()) {
  tell_object (this_player(), "If this were an active quest, you would have "
    "just gotten a wodge of XP!\n");
}
```

If we want to make this quest operational, we replace this with a call to set_quest in the library. This marks a quest as being completed by the given player.

```
if (check_complete()) {
  LIBRARY->set_quest (this_player()->query_name(), QUEST_NAME);
}
```

Now, let's talk about the last bit of functionality we need to add to our quest to make it fully functioning.

Cast Your Mind Back...

The last bit of the quest was that it was supposed to award the player with a secret code that worked on the secret door to come. That requires us to have some method of storing information on our player in a portable, easily accessible way.

One method we have is the use of properties. We don't want to do that. Properties are useful, quick and easy – but they tend to linger around and people forget what they are for. Unless there's a compelling reason to use a property, we should avoid it.

One of the things that the library gives us is a mechanism for storing quest-related information along with players. This information gets stored as a mixed variable, and we have the responsibility in our code for working out what kind of data it should be, and parsing it appropriately. If all we're storing is a single piece of information, it's fine – otherwise we need to put some code in around the edges. We'll come back to that later...

We can set player information using the method set_player_quest_info method on the library. So lets generate a random code for a player, and have it stored along with the player's name:

```
string generate_random_code() {
  int rand = random (9000);
  rand += 1000;
  return "" + rand;
}
```

And

```
if (check_complete()) {
  LIBRARY->set_quest (this_player()->query_name(), QUEST_NAME);
  LIBRARY->set_player_quest_info (this_player()->query_name(),
    QUEST_NAME, generate_random_code());
}
```

We can also pull information out with query_player_quest_info. Normally we'd store this information in the function rather than querying it from the library, but this is just proof of concept:

```
if (check_complete()) {
  LIBRARY->set_quest (this_player()->query_name(), QUEST_NAME);
  LIBRARY->set_player_quest_info (this_player()->query_name(),
    QUEST_NAME, generate_random_code());
  code = LIBRARY->query_player_quest_info (this_player()->query_name(),
    QUEST_NAME);

  tell_object (this_player(), "As you put away the last book, a small slip "
    "of aged paper flutters to the ground.  You can see it has the "
    "numbers \"" + code + "\" on it before it crumbles into dust.\n");
}
```

Now you get a new piece of information when the quest completes:

```
As you put away the last book, a small slip of aged paper flutters to the
ground.  You can see it has the numbers "2023" on it before it crumbles into
dust.  You sort the book "Some Stuff About You" into category "Romance"
```

This is the information that we need for the later quest, now happily stored and ready for us to make use of later. This allows us to handle things like storing quest stage data – for example, as a player progresses through a quest, an integer value can be stored to represent what state the quest is in for that specific player.

Finally, we can put restrictions on participation with completed quests through the use of the query_quest_done method, like so:

```
if (LIBRARY->query_quest_done (this_player()->query_name(), QUEST_NAME)) {
  add_failed_mess ("You have already done this quest.  Give someone else a "
    "chance..\n", ({ }));
  return 0;
}
```

The combination of these four methods will allow you to manage the functionality for multi-stage quests. However, it's a little clunky, and so our mudlib has an option for making the syntax a little less painful to work with.

Quest Utils

In our mudlib is an object called /obj/misc/quest_info_utils. It's defined in library.h as QUEST_UTILS. You can inherit this along with the rest of your inherits – all it does is give you an easier way of interacting with the library. However, there is a limitation in that it can only be used in an object if that object requires to interact with the library for one quest only. Luckily, that's almost all of our objects, so it's not much of a drawback. It is however a problem for our development here.

First of all, in the setup of our object, we need to make a call to set_quest_name:

```
set_quest_name (QUEST_NAME);
```

Once this is done, we no longer have to go through the library to update player details. Most usefully, the quest utils object manages player quest data as a mapping, so manging player state information for quests is reduced to simply setting key and value pairs. We use the method set_quest_param to do this:

```
if (check_complete()) {
  set_quest (this_player());
  set_quest_param (this_player(), "random code", generate_random_code());

  code = query_quest_param (this_player(), "random code");

  tell_object (this_player(), "As you put away the last book, a small slip "
    "of aged paper flutters to the ground.  You can see it has the "
    "numbers \"" + code + "\" on it before it crumbles into dust.\n");
}
```

We also have our query_quest_done call simplified like so:

```
if (query_quest_done (this_player())) {
  add_failed_mess ("You have already done this quest.  Give someone else a "
    "chance.\n", ({ }));
  return 0;
}
```

All the quest utilities do are simplify the syntax – which of these methods you choose to use will depend on your own individual needs as well as which syntax you feel most comfortable with. If at any point you have an object that requires you to manipulate the quest data for multiple quests, you'll have to go with what we might call the ‘longhand' version. Alas, we need to do that for this particular room, because there are two quests here.

Our Second Quest

Let's move on from this quest on to our second quest – the hole in the roof that leads us on to the second floor of the library. Remember how we planned this quest out in Being A Better Creator. We have a dynamic setup that requires us to go through several steps:

1 Randomly generate a location for the hole 1 If the player has a certain other.perception bonus, show the hole when they look in the right place. 1 When a player looks at the hole, give them the clues as to how they need to break through it. 1 The hole is reached by the player climbing the appropriate bookshelf. 1 The player uses the hints given as to tool needed to break through the hole. 1 Upon doing so, the quest is granted and the player is moved into the secret room above.

So, now we need to first generate a random location for a hole. Ideally, we want to reach it by climbing up a particular category bookshelf. So, let's do this – when the player looks at the categories or the bookshelves, we want to do a perception check and then generate a location for them to see the hole. We also want to generate a random tool to break through the roof. We'll store these as an array and then set the player's quest info appropriately:

```
string* check_hole_location (object player) {
  string *details = LIBRARY->query_player_quest_info (player->query_name(),
    HOLE_QUEST);
  string *cat = ({"romance", "action", "childrens"});
  string *tools = ({"crowbar", "hammer"});

  string hole;
  string tool;

  if (details && sizeof (details) == 2) {
    return details;
  }

  switch(TASKER->perform_task(player, "other.perception", 120, TM_FREE ) ) {
    case AWARD :
      tell_object( player, "%^YELLOW%^%^BOLD%^You feel a little more "
        "perceptive.%^RESET%^\n" );
    case SUCCEED :
      hole = element_of (cat);
      tool = element_of (tools);
      details = ({hole, tool});
      LIBRARY->set_player_quest_info (player->query_name(), HOLE_QUEST,
        details);
    break;
    default :
      hole = 0;
  }
  return details;
}
```

We hook this into our shelf add_item so that people looking at the add_item have a chance of seeing the hole:

```
string books_on_shelves() {
  string ret = "The bookshelves are arranged according to categories.  ";
  string *details;

  details = check_hole_location (this_player());

  if (!sorting || !sizeof (sorting)) {
    ret += "No books have been sorted onto the shelves.";
  }
  else {
    foreach (string book, string category in sorting) {
      ret += "The book \"" + cap_words (book)
        + "\" has been put on the shelf for the category \"" +
        cap_words (category) + "\"\n";
    }
  }

  if (details) {
    ret += "  You can see a crack in the roof above the \"" + details[0]
      + "\" bookshelf.  If you had a " + details[1]
      + ", you could probably break through the roof.";
  }

  return ret;
}
```

Lovely, huh? Now we need an add_command that lets the player break the roof:

```
add_command ("break", "roof above <string'category'> bookshelf");
```

And then the code to handle it:

```
int do_break (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string category = args[0];
  string *details;
  string location, tool;
  object *valid_tools;

  if (LIBRARY->query_quest_done (this_player()->query_name(), HOLE_QUEST)) {
    add_failed_mess ("You have already broken through the roof, no need "
      "to do it again.\n", ({ }));
    return 0;
  }

  details = check_hole_location (this_player());

  if (!details) {
```

```
    add_failed_mess ("Break what?  Why?  You are a bad person for "
      "being such a vandal.\n", ({ }));
    return 0;
  }

  location = details[0];
  tool = details[1];

  if (category != location) {
    add_failed_mess ("The roof there is too solid to break.\n", ({ }));
    return 0;
  }

  valid_tools = match_objects_for_existence (tool, this_player(),
    this_player());

  if (!sizeof (valid_tools)) {
    add_failed_mess ("You have no tool suitable for breaking through "
      "the roof.\n", ({ }));
    return 0;
  }

  LIBRARY->set_quest (this_player()->query_name(), HOLE_QUEST);

  add_succeeded_mess ("$N break$s through the roof with $I.\n",
    ({ valid_tools[0] }));

  return 1;
}
```

Our final step is then adding the command that lets the player climb up through the hole. So that they don't need to keep breaking it, we'll make it conditional on them having actually completed the quest previously. However, since we're not actually completing these quests (they are set to inactive so they don't show up in player scores), we need to comment that bit out to test it.

And there we have it – one room, two quests – it's tremendous value for money by any standard!

Conclusion

We've made a lot of progress in this chapter – we turned our quest architecture into an actual quest, and then added in the second of our three quests. Okay, so the second one isn't especially interesting, but it's there!

We have one more quest to add to our development, and then we have our three. Then there are all sorts of other exciting things we need to include in our village. Are you excited? I'm excited – touch your nose if you're excited too!

Our Last Quest

Introduction

Two quests behind us – we're burning through our list of features here! Now we add in the third, and then we can devote our attention to the other parts of the village we are currently ignoring. Our last quest as I am sure you can recall from Being A Better Creator is to find a secret panel behind a painting, and then enter the secret code that we were given to open a secret door. The clues that we give to this quest should be provided by the sorted library below – we should be able to ‘research' details about the portraits and the hints that we get reveal the answer to the puzzle.

That's the topic for this chapter – researching in the library, and the mechanisms of pushing portraits.

The Quest Design

The second floor of our library is going to be a portrait gallery, and behind one of these portraits is a secret panel. Finding the portrait will be the quest, rather than simply entering the code – after all, that's just ferrying a number from one place to the other.

The puzzle we build then should be logical, possible to solve by diligent effort, and that effort should be a worthwhile shortcut to simply brute-forcing the quest by pushing aside every single portrait. Our design then must accommodate this.

First, let's think about how to handle the brute-forcing bit. We have two obvious courses:

- Thousands of portraits
- A limited number of guesses

Thousands of portraits could be generated fairly easily by a function, but it seems like a pretty awkward solution. On the other hand, if we make it so they can push aside perhaps three portraits in twenty before the state of the puzzle resets, then it's important that those three portraits are the right, or sensible portraits.

Let's consider what these portraits might actually be – it's a wizard's tower, so they are likely to be pictures of wizards rather than aristocratic relatives. It's pretty easy to algorithmically generate twenty or so suitably wizardly names. Likewise, we can decide on some features that define each one – size of head, colour of hair, height, weight, eye-colour, attractiveness, and so on. Then, we can provide a set of clues that uniquely identify one specific portrait as being worth moving.

We could hard-code each of these portraits, but we like dynamic quests here on Discworld, so we'll do them randomly.

Anyway, enough of that though, we are eager for action! But before we begin coding, let's talk about a very Discworld Mud way of dealing with distributed data access.

Handlers

We use the word ‘handler' a lot on Discworld. For new creators, they often conjure up mental images of horribly complicated and important code, locked away in mysterious mudlib directories and not for casual perusal. While that's true of a number of our handlers, the basic concept is very simple.

In our scenario above, we are faced with the task of co-ordinating some functionality across two separate rooms. We must be able to research people in the library, and the people we research must be represented in portrait form on the second floor. We need a way of storing data so that these two objects can have access to it.

One way to do this is to have one room call functions in another room. Unfortunately, due to the way rooms work on Discworld, this is not a reliable strategy. Rooms unload themselves when no-one is in them so that we reduce memory load. When a function gets called on an unloaded room, it forces the room to reload – along with any setup that gets done. So if we generate twenty random portraits, then simply moving from one room to another may be enough to change the entire state of the quest for a player. They go from a room containing one set of portraits to a library referring to another set. That's not good.

Instead, what we use is a handler – a handler is just a normal object that is designed to be accessed by multiple other objects. It defines a set of data and stores it in a single place, and it provides functions for acting on that data. Henceforth, no one object has to store this data because they just make calls on the handler. Really, the kind of handlers that are associated with small area developments are mini versions of the more substantial handlers that exist over entire domains and in /obj/handlers. Nonetheless, the concept remains the same.

There is often some confusion as to the distinction between a handler and an inherit – they do after all seem to occupy similar roles in coding. The difference lies in the persistence of data – in an inherit, each object gets its own copy of the data, and if that data is randomly set up, it'll have a different set of data from all the other objects. In a handler, there is only one set of data and all objects make use of that single set. In terms of what inherits allow you to do with shared functionality, they are almost identical except that handlers are easier to add in after the fact.

Our taskmaster is a handler. There is no reason why things like perform_task couldn't be provided as a function as part of /std/object. However, if you were writing a piece of code that didn't inherit /std/object, then you'd need to bind in the inherit yourself. Additionally, if you wanted to change the way perform_task worked, you'd need to force a reboot of the MUD – otherwise you couldn't be sure which version of the perform_task code any given object would be using at any time.

In essence, you use an inherit when all you care about is a set of objects having access to some common set of functionality. You use a handler when you want a set of objects having access to some common set of data. For our purposes, we need a handler.

The Portrait Handler

Our handler is not complicated. It doesn't inherit from anything because it's just an object for storing some data and functionality. We'll put it in a new sub-directory of Betterville, and include a new define in our path.h:

```
#define   INHERITS        BETTERVILLE + "inherits/"
#define   ROOMS           BETTERVILLE + "rooms/"
#define   HANDLERS        BETTERVILLE + "handlers/"
```

It just needs a create method – much like the first look we had at the code for the library:

```
void create() {
  seteuid (geteuid());
}
```

Now, we build a data representation. Since we've already done some work with classes, let's make a class for a portrait:

```
class portrait {
  string name;
  string title;
  string eyes;
  string height;
  string weight;
  string looks;
  string hair;
}

class portrait *all_portraits;

void create() {
  seteuid (geteuid());
  all_portraits = ({ });
}
```

Now we need something to randomly generate each of the different elements of the portrait. A real quest would be more imaginative in the names we apply here, but never mind. First, something to generate a random name:

```
string random_wizard_name() {
  string *forename = ({"albert", "mustrum", "ponder", "bill", "achmed",
    "drum", "eric", "eskarina", "harry"});
  string *surname = ({"malicho", "ridcully", "stibbons", "rincewind",
    "billet", "smith", "dread"});

  return element_of(forename) + " " + element_of(surname);
}
```

And something to generate a random title:

```
string random_wizard_title() {
  int level = 4 + random (5);
  string *orders = ({
    "The Ancient and Truly Original Sages of the Unbroken Circle",
    "The Ancient Order of the Dynastic Crescent",
    "The Ancient Order of Djinn Diviners",
    "The Hoodwinkers",
    "The Last Order",
    "Mrs. Widgery's Lodgers",
    "The Order of Midnight",
    "The Ancient Order of the Scintillating Scarab",
    "The Venerable Council of Seers",
    "The Sages of the Unknown Shadow",
    "The Ancient and Truly Original Brothers of the Silver Star"
  });

  return "the " + word_ordinal (level) + " level wizard of "
    + element_of (orders);
}
```

The rest we can generate a little more easily. However, we need the name of the wizard to be a unique identifier – how else will we be able to allow players to research them? So we first need a way to find a specific wizard's portrait. We've already seen this in action:

```
int find_portrait (string name) {
  for (int i = 0; i < sizeof (all_portraits); i++) {
    if (all_portraits[i]->name == name) {
      return i;
    }
  }
  return -1;
}
```

Now, let's make a method that creates an entirely random portrait:

```
void add_portrait() {
  class portrait new_portrait;
  string wname;
  string wtitle;
  string *eyes_arr = ({"blue", "green", "black", "red"});
  string *height_arr = ({"tall", "short", "medium-height"});
  string *weight_arr = ({"fat", "skinny", "stocky", "plump"});
  string *looks_arr = ({"handsome", "ugly", "grotesque", "beautiful"});
  string *hair_arr = ({"black", "blonde", "grey", "white", "brown"});

  do {
    wname = random_wizard_name();
  } while (find_portrait (wname) != -1);

  wtitle = random_wizard_title();

  new_portrait = new (class portrait,
    name: wname,
    title: wtitle,
    eyes: element_of (eyes_arr),
    height: element_of (height_arr),
    weight: element_of (weight_arr),
    looks: element_of (looks_arr),
    hair: element_of (hair_arr));

  all_portraits += ({ new_portrait });
}
```

Then we add a method that gives us the string description of a portrait (what we get when we look at it). We're going to use an efun called **sprintf** here – sprintf lets us perform some sophisticated string parsing without us having to play around with concatenation and such. It works very similarly to a pattern in an add_command, except we use a %s to represent a string and we pass the bits to be filled in as parameters to the function, like so:

```
string portrait_long (string wizard) {
  int i = find_portrait (wizard);

  if ( i == -1) {
    return 0;
  }

  return sprintf ("This is a portrait of %s, %s.  He is a %s, %s man, "
    "with %s hair, %s eyes and a %s face.\n",
    cap_words (all_portraits[i]->name), all_portraits[i]->title,
    all_portraits[i]->height, all_portraits[i]->weight,
    all_portraits[i]->hair, all_portraits[i]->eyes, all_portraits[i]->looks);
}
```

When we call this function with a wizard represented as a portrait, we get something like the following back out:

```
This is a portrait of Harry Stibbons, the eighth level wizard of The Ancient
Order of the Scintillating   Scarab.  He is a tall, fat man, with grey hair,
red eyes and a beautiful face.
```

Now we just need a mechanism to identify which of these portraits is the one behind which the secret panel may be found. Exactly how we'd choose to do this in a real quest for the game doesn't matter – we don't want to lock great quest ideas away in a tutorial! Aside from the library quest (which would be serviceable as an actual quest for the game with some modification), these quests are illustrations of concept rather than quests we'd expect people to enjoy.

What we're going to do next is illustrate why classes are such a good way of representing data. We need to add in a new element of the class – a ‘history' for people to find when they research the wizard. We'll assign these semi-randomly by means of a parameter passed to the add_portrait method:

Step one is to add two new elements to the class:

```
class portrait {
  string name;
  string title;
  string eyes;
  string height;
  string weight;
  string looks;
  string hair;
  string history;
  int secret_panel;
}
```

And then a modification to add_portrait:

```
void add_portrait(int suspected) {
  class portrait new_portrait;
  string wname;
  string wtitle;
  string *eyes_arr = ({"blue", "green", "black", "red"});
  string *height_arr = ({"tall", "short", "medium-height"});
  string *weight_arr = ({"fat", "skinny", "stocky", "plump"});
  string *looks_arr = ({"handsome", "ugly", "grotesque", "beautiful"});
  string *hair_arr = ({"black", "blonde", "grey", "white", "brown"});
  string *banal_history = ({
    "he was a great battle wizard in the employ of the dwarfs at "
      "Koom Valley.",
    "he was a great archchancellor of Unseen University.",
    "he was thoroughly unremarkable in every way.",
    "he had no distinguishing accomplishments."
  });

  string *suspect_history = ({
    "he was an especially cunning and secretive researcher.",
    "he had a gift for traps, secret doors, and mechanical constructions.",
    "he was known for hiding things in obvious places.",
  });

  string hist;

  if (suspected) {
    hist = element_of (suspect_history);
  }
  else {
    hist = element_of (banal_history);
  }

  do {
    wname = random_wizard_name();
  } while (find_portrait (wname) != -1);

  wtitle = random_wizard_title();

  new_portrait = new (class portrait,
    name: wname,
    title: wtitle,
    eyes: element_of (eyes_arr),
    height: element_of (height_arr),
    weight: element_of (weight_arr),
    looks: element_of (looks_arr),
    hair: element_of (hair_arr),
    history: hist,
    secret_panel: suspected);

  all_portraits += ({ new_portrait });
}
```

Now, if we pass in a positive number as a parameter, we'll get one of the ‘suspect' histories. We'll make it so that a secret_panel value of 1 means that the person **might** be where the panel is, and 2 means that's where it **actually** is. Next, we need a method to setup the state of the handler:

```
void setup_data() {
  int location = random (10);
  int panel;

  all_portraits = ({ });

  for (int i = 0; i < 10; i++) {
    panel = 0;

    if (i == location) {
      panel = 2;
    }

    else if (i % 3 == 0) {
      panel = 1;
    }

    add_portrait (panel);
  }
}
```

So here we generate where the panel is going to be, and we make every third portrait a ‘suspect' one. Now, although we can query the handler as to which portrait is which, every time the method is called we get a brand new set of data to work with. Before we're done here though, we need to add in a few more utility functions:

```
string query_portrait_history (string wizard) {
  int i = find_portrait (wizard);

  if ( i == -1) {
    return 0;
  }

  return all_portraits[i]->history;
}

int query_portrait_panel (string wizard) {
  int i = find_portrait (wizard);

  if ( i == -1) {
    return 0;
  }

  return all_portraits[i]->secret_panel;
}

string* query_portraits () {
  string *names = ({ });

  for (int i = 0; i < sizeof (all_portraits); i++) {
    names += ({ all_portraits[i]->name });
  }

  return names;
}
```

Back To The Library

So, we're not done with our library yet. Now we're going to add yet another piece of functionality – the ability to ‘research' things. Some of these things will be related to other parts of the village, some should be related to other players (as per our village design in Being A Better Creator), but we also want people to be able to research wizards to see what they are known for.

The process for doing this should be familiar now – we create an add_command. However, instead of making use of functions in our room, we're going to make calls on the handler we just created.

Handlers are usually #defined in a header file somewhere. We have a define for our handlers directory, but we also want one for the handler we created itself:

```
#define   INHERITS        BETTERVILLE + "inherits/"
#define   ROOMS           BETTERVILLE + "rooms/"
#define   HANDLERS        BETTERVILLE + "handlers/"
#define   PORTRAIT_HANDLER   (HANDLERS + "portrait_handler")
```

Notice here that we surround the path in brackets, which we haven't done before. The reason for this is the pre-processor that does all the clever replacing of our defines in our code. We'll come back to this.

We need an add_command as usual;

```
add_command ("research", "<string>");
```

And the code to handle it:

```
int do_research (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string history;
  string topic;

  topic = args[0];

  if (check_complete()) {
    add_failed_mess ("The library is too disorganised for you to be able "
      "to find any useful information.\n", ({ }));
    return 0;
  }

  history = PORTRAIT_HANDLER->query_portrait_history (topic);

  if (!history) {
    add_failed_mess ("The library doesn't seem to have any information on "
      "that topic.\n", ({ }));
    return 0;
  }

  tell_object (this_player(), "As far as you can tell, "
    + history + "\n");

  add_succeeded_mess ("$N flick$s through some of the books "
    "in the library.\n", ({ }));
  return 1;
}
```

Note how we make our call on the portrait handler in exactly the way we've made calls on the taskmaster in the past. This is why we need to use the brackets in our define. The pre-processor does the search and replace on the code we write, putting in the correct values for each of our defines. If we miss out the brackets, this is what the code looks like to the compiler:

```
history = "/d/learning/betterville/" + "handlers/" + "portrait_handler"-
>query_portrait_history (topic);
```

Due to the way the MUD evaluates these operators, it attempts to call the function on the object “portrait_handler", rather than the fully qualified name. With the brackets, the code looks like this:

```
history = ("/d/learning/betterville/" + "handlers/" + "portrait_handler")-
>query_portrait_history (topic);
```

The brackets tell the driver ‘Hey, put all this together before you attempt to call that function'. This ensures the function gets called on the fully qualified "/d/learning/betterville/handlers/portrait_handler", which is an actual object it knows how to find.

Conclusion

We only have the second floor of the library to do now to make this quest work, and that follows on naturally from what we've been discussing in this chapter. We've got a handler now, and being able to bundle all the functionality into that greatly simplifies the task of implementing the rest of this quest. Handlers are a common and powerful approach to development on Discworld, and you'll find that you will have cause to make use of them often if developing code a little beyond the ordinary.

They are nothing to be feared – they are identical to almost every other piece of code you have seen, and you are very much capable at this point of developing your own, as we have just done.

Finishing Touches

Introduction

And now we round off our fairly lengthy discussion of the Betterville quests with the last part of the puzzle – the second level of the library. In this room we tie together all of our quests into a neat, integrated whole. We've already got a portrait handler, so now all we need to do is provide the mechanism for moving portraits aside and entering the secret code. Come, take my hand – I have such wonders to show you.

The Second Level Room

Okay, let's start off with the room itself. It's a simple, humble room – but it's ours. We begin with a skeleton:

```
#include "path.h"
#include <library.h>

inherit INHERITS + "inside_room";

void setup() {
  set_short ("second level of the library");
  add_property ("determinate", "the ");
  set_long ("This is the second level of the Betterville library.  It's "
    "full of portraits of crazy looking wizards.\n");
  set_light (100);
  add_exit ("down", ROOMS + "main_library", "path");
}
```

And there's a pretty familiar pattern we follow. First, we add in some add_items that let us represent the view of the data from our handler. One for portraits to begin with, so that our players can see the portraits available for them to view:

```
string portrait_long() {
  string *portraits = PORTRAIT_HANDLER->query_portraits();
  return "There are many portraits hanging around the room.  You can see "
    "portraits for the following wizards: " + query_multiple_short
    (portraits) + ".  Undoubtedly more information about each of them "
    "could be researched in the library.\n";
}
```

Now when we look at the portraits, we'll see something like this:

```
There are many portraits hanging around the room.  You can see portraits for
the following wizards: albert malicho, ponder   billet, eskarina billet,
achmed   stibbons, drum ridcully, albert ridcully, eric malicho, eric
stibbons, eskarina   malicho and eric smith.
```

Unfortunately, we need to do a little bit of processing so that it actually looks good – capitalising each of the names. There are ways we can do this quickly and easily, but we'll have to do it longhand for now:

```
string portrait_long() {
  string *portraits = PORTRAIT_HANDLER->query_portraits();

  for (int i = 0; i < sizeof (portraits); i++) {
    portraits[i] = cap_words (portraits[i]);
  }

  return "There are many portraits hanging around the room.  You can see "
    "portraits for the following wizards: " + query_multiple_short
    (portraits) + ".  Undoubtedly more information about each of "
    "them could be researched in the library.\n";
}
```

We've already handled the research part, so now we just need to add in the mechanism for handling the quest. Remember, we've decided upon there being a limit to the number of guesses a player can make regarding the portraits they work with – let's set it at three guesses. Each wrong guess will result in a clue to the player, and a correct guess will reveal the secret pad for them to enter the number.

I'm Looking Through... uh... behind you

Looking behind a portrait is a pretty straight-forward affair – an add_command does the business as ever:

```
add_command ("look", "behind portrait of <string>");
```

And then we need the function to handle it. Remember that we decided that the secret_panel value of the portrait would define if it was where the number-pad could be found:

```
int do_look (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string wizard;
  int panel;
  int wiz;
```

```
  wizard = lower_case (args[0]);

  wiz = PORTRAIT_HANDLER->find_portrait (wizard);

  if (wiz == -1) {
    add_failed_mess ("You can't find a portrait of that wizard...\n",
      ({ }));
    return 0;
  }

  panel = PORTRAIT_HANDLER->query_portrait_panel (wizard);

  if (panel == 2) {
    tell_object (this_player(), "There is a number-pad behind the "
      "portrait of " + cap_words (wizard) + "!  It's just waiting "
      "for you to enter a number.\n");
  }

  add_succeeded_mess ("$N move$s aside a picture and look$s behind it.\n",
    ({ }));

  return 1;
}
```

Now we need to decide how to handle the finding of the secret panel. We could create an actual secret panel object that gets cloned into the room when a player finds it – that would work. Or, we could simply update the player's quest info with the appropriate stage of the quest. Either of these is a workable solution, but we're going to go for the latter because it removes a number of complications (for example, should a player that wanders into a room after another player be able to see the secret number pad?). We'll make entering the code handled by another add_command, and make the entrance criteria of that command linked to the quest stage the player is at. Simple!

This quest in the quest handler is stored as ‘betterville secret door', so that's the quest we're going to define. Remember too that we need access to the quest info for ‘betterville library' in order to determine if the code entered is correct - we can't use the query_info_utils object here either. So, we define our door quest as DOOR_QUEST, and incorporate an update of the quest info in the success:

```
if (panel == 2) {
  tell_object (this_player(), "There is a number-pad behind the portrait "
    "of " + cap_words (wizard) + "!  It's just waiting for you to enter "
    "a number.\n");
  LIBRARY->set_player_quest_info (this_player()->query_name(),
    DOOR_QUEST, 1);
}
```

That's the portrait side handled – most of the work is done in the handler, so we just need to provide methods to query the various elements. Next, we move on to handling the secret code.

Ecretsay Odecay

To begin with, the add_command:

```
add_command ("enter", "code <string>");
```

And then the functions to handle it:

```
int do_enter (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {

  string entered_code = args[0];
  string needed_code;

  if (!LIBRARY->query_quest_done (this_player()->query_name(),
    DOOR_QUEST)) {

    add_failed_mess ("Enter what into what?  You're crazy, dude.\n",
      ({ }));

    return 0;
  }

  needed_code = LIBRARY->query_player_quest_info
    (this_player()->query_name(), LIBRARY_QUEST);

  if (entered_code != needed_code) {
    add_succeeded_mess ("$N enter$s a code on a secret number pad, but "
      "nothing happens.\n", ({ }));
    return 1;
  }

  add_succeeded_mess ("$N enter$s a code on a secret number pad.\n",
    ({ }));
  call_out ("transport_player", 0, this_player());

  return 1;
}

void transport_player (object player) {
  tell_object (player, "The world goes black for an instant, and when "
    "you open your eyes you find yourself somewhere new...\n");
  player->move_with_look (ROOMS + "library_third_level",
    "$N appear$s with a flash!",
    "$N disappear$s with a flash of light!");
}
```

Note the use of the call_out in the do_enter method – unfortunately, the nature of message parsing on the MUD means that it's sometimes difficult to enforce the proper order in which we want messages to be printed. The call_out ensures that our function has time to complete before our new messages get shown.

Random Guessing

We noted in an earlier chapter that we didn't want players to simply look behind every single portrait in order to find the secret pad – we want them to have a limited number of guesses. That's pretty easy to handle – in the room, we hold a mapping that stores how many guesses a player has had. We can clear it every time reset is called:

```
mapping guesses;

void reset() {
  guesses = ([ ]);
}
```

Then in the do_look method, we incorporate the code for handling the number of guesses along with a message indicating how close the player has come to using up their last chance. Of course, these messages are just proof of concept – real quests don't have quite such feeble justification. Or at least, they shouldn't!

```
num_guesses = guesses[this_player()->query_name()];

num_guesses += 1;

guesses[this_player()->query_name()] = num_guesses;

if (num_guesses > 3) {
  add_failed_mess ("None of the portraits will move for you!  How odd.\n",
    ({ }));
  return 0;
}

switch (num_guesses) {
  case 1:
    tell_room (this_object(), "The painting moves easily, but there is "
      "an ominous glow that appears and then fades away...\n");
    break;
  case 2:
    tell_room (this_object(), "It is hard to move the portrait - as if "
      "someone was holding on from the other side of the frame.  Odd.\n");
  break;
  case 3:
    tell_room (this_object(), "It is almost impossible to move the "
      "portrait.  It's like it has suddenly become magically glued to "
      "the wall!\n");
  break;
}
```

Now each player gets a maximum of three attempts to find the secret panel per reset. If we want to be Complete Bastards, we can even make it so that the entire state of the quest is set up anew every reset:

```
void reset() {
  guesses = ([ ]);

  PORTRAIT_HANDLER->setup_data();
  tell_room (this_object(), "There is a strange flash, and all the "
    "portraits in the room morph and change into new and interesting "
    "shapes and combinations.\n");
}
```

And that's it – all three quests in place. They are crying out for polishing and prettying up, but the process of building the quests is complete. There is always more we can do, and interested creators are advised to spend time looking at how they can adapt these quests to be better and more interesting – it's always a valuable lesson to expand on a framework that is already provided.

What Kind Of Day Has It Been?

So, what have we learned over the past few chapters? Well, quite a lot! However, what we haven't learned really is to write quests. What we've learned is how to write these specific quests, and use the tools that are available to build a back-end architecture and a user front-end. Every quest is different though – like delicate little snow-flakes. For each one, you'll need to go through the same process, but the results will always be different:

- Choose a data representation
- Implement methods for managing that representation
- Put in place a user interface for those methods

The important thing to take away from these last chapters is the technical tools we have been discussing:

- classes
- inheritables
- add_commands
- handlers
- the quest library

These will serve you in good stead for putting together the back-end of any quest you can imagine. The limit is really your imagination. The quests that we have put in place have shown us examples of all of these tools being used in concert to achieve a fairly complex end.

Conclusion

Three quests – that's not to be sniffed it. Fair enough, they're not particularly good quests, and lack a certain panache – but we'd be doing ourselves a disservice if we hid all our best ideas in a set of tutorial documents. Nonetheless, what we have are three operational quests involving dynamic quest design and a consistent narrative. Understanding how these three quests are put together is the key to developing your own original and much more interesting developments.

Now that we have finished with the library (for now), we'll move on to the other parts of the village. We've still got a Beast to write, some NPCs, and a shop full of merchandise for gullible gold-diggers! Our village demands our attention, we must answer its siren call!

Gold Diggers

Introduction

We've been through some pretty intense material in the past few chapters, so let's calm the pace a little by talking about our cannon fodder – the dozy girls who make up the majority of the wandering population of Betterville. They have been lured to the village by distorted tales of a handsome prince looking for love's true kiss to restore him to his normal form.

We know how to develop NPCs – we did that several times as part of LPC For Dummies 1. In this chapter we're going to extend the concept a little to discuss some of the additional functionality built into NPC objects.

The Dozy-Girl Template

Because our dozy girls are the only wandering NPCs in the village, we want them to look as different as possible. As such, we won't develop NPCs with static descriptions – they'll be dynamically generated to ensure variety. So, we create a chars directory in Betterville and add it to our path.h:

```
#define   INHERITS        BETTERVILLE + "inherits/"
#define   ROOMS           BETTERVILLE + "rooms/"
#define   HANDLERS        BETTERVILLE + "handlers/"
#define   CHARS           BETTERVILLE + "chars/"
#define   PORTRAIT_HANDLER   (HANDLERS + "portrait_handler")
```

In our dozy-girl NPC, we're going to make use of a lot of things we've discussed in previous chapters to add the dynamics. It's not a complex system, but it's very flexible. Let's start off simply with the basic framework. Let's create an NPC inherit, just because we can:

```
inherit "/obj/monster";

void create(){
  do_setup++;
  ::create();
  do_setup--;

  if ( !do_setup ){
    this_object()->setup();
    this_object()->reset();
  }
}
```

We'll use this inherit for all our Betterville NPCs, just in case we want to add any common functionality to the various inhabitants of the area.

Setting up the NPC involves us making use of some randomised elements – we want to set the long description based on the location the girl is from, and their name and short based on their type Thus, we need to set up a pair of arrays to hold the possible choices:

```
string *types = ({"gold digger", "wannabe princess",    "upwardly mobile
socialite"});
string *places = ({"Lancre Town", "Genua"});
string place = element_of (places);
string type = element_of (types);
```

We first want to set the name of the NPC – the actual name itself will be the last word in the type (so for ‘wannabe princess' the name should be ‘princess'), and the rest of the parts of the type will be adjectives. First then, we make an array out of the type of NPC, which we can do by using the **explode** efun:

```
string *arr;
arr = explode (type, " ");
```

We can get the last element of the array by using the <1 indexing system:

```
name = arr[<1];
```

Then we subtract the last element from the array:

```
arr -= ({ arr[<1] });
```

Having done this, we can actually setup the various parts of our NPC:

```
set_name (name);

foreach (string a in arr) {
  add_adjective (a);
}

set_short (type);
basic_setup ("human", "warrior", 25 + random (25));
set_gender (2);

switch (place) {
  case "Lancre Town":
    str += "This is a girl from Lancre Town.  Lancre has few opportunities "
      "for social mobility ever since Verence married Magrat, and so she "
      "has come here hoping to improve her lot in life.  ";
   break;
   case "Genua":
     str += "This is a girl from Genua.  Genua City is full of "
       "opportunities for girls looking to marry into wealth, but "



       "this is done in a cutthroat environment of vicious "
       "competition.  She has come to Betterville in the hope "
       "of finding an easier time of it in the Provinces.  ";
    break;
  }
  set_long (str);
```

Next, we can make use of the same system that we used for the portraits to describe each of the girls. First we declare arrays to hold all the choices:

```
string *lengths = ({"long", "short"});
string *colours = ({"blonde", "brunette", "red", "black"});
string *weight = ({"fat", "thin", "skeletal", "plump", "well-built"});
string *height = ({"tall", "short"});
```

We are going to handle the building of the descriptive string a little differently here – we won't use sprintf (although it is a tremendously lovely little function). Instead we're going to create our own control codes and use the **replace** efun on the description:

```
str += "She is a $height$, $weight$ girl wih $length$ $colour$ hair.\n";

str = replace (str, ({
  "$height$", element_of (heights),
  "$weight$", element_of (weights),
  "$length$", element_of (lengths),
  "$colour$", element_of (colours),
}));

set_long (str);
```

By using the replace efun, we can thread these random elements easily throughout a whole long description. For example, if we added in a new origin for the dozy girl, we could make use of it in the description string like so:

```
case "Sto Lat":
  str += "This is a girl from Sto Lat.  In the city of Sto Lat, $weight$ "
    "girls like this are not especially sought after by the courtiers, "
    "and so she has come here to find love with someone easier to "
    "please.  Or at least, easier to please for someone of her gender.  ";
break;
```

The sprintf function doesn't allow us to easily thread these kind of details anywhere in a string but this more flexible replace system does – both approaches are perfectly valid, you should choose which is most appropriate for your actual needs. Finally, we need to set up our girls with some load_chats and nationalities. We setup the nationalities in our switch statements, and include the load_chats as usual. As an example of a nationality:

```
case "Lancre Town":
  str += "This is a girl from Lancre Town.  Lancre has few opportunities "
    "for social mobility ever since Verence married Magrat, and so she "
    "has come here hoping to improve her lot in life.  ";
  setup_nationality ("/std/nationality/lancre", "Lancre Town");
break;
```

And then our load_chats:

```
load_chat(120,  ({
  1, "' I could be the one his One True Love, I know it!",
  1, "' I have travelled far to get here, but love awaits me.",
  1, "' I wonder what I'll spend his vast fortune on first when "
      "I have broken the curse...",
  1, "' Have you seen the Beast?  I'm going to be his one true love.",
  1, "' They say a kiss will turn him back into a prince.",
  1, "' We're going to get married in the spring. You know, when "
     "I find him.",
  1, "' All these other girls are fools if they think they are his One."
}));

load_a_chat(120,  ({
  1, "' Help, my Love will save me!",
  1, "' I don't remember this part of the fairy-tale!",
  1, "' The Beast won't be happy you're killing his one true love!",
  1, "' Stop, this is part of a different story!",
  1, "' I think we all know who the real beast here is!"
}));
```

We'll dress them up later, once we've written the shop in the village square. For now, let's move on to more interesting territory.

Event Driven Programming

One of the things that the MUD does at pre-set intervals is generate **events**. These are functions that get called on objects when certain things have happened in the game. We can provide functions that catch these events and execute code when the events occur. This is something easier to see in practise than it is to describe in theory, so let's look at a specific example.

One of the times that the MUD triggers an event is when an NPC or a player dies. A function gets called on the NPC itself, as well as all of the objects in the environment of the NPC. The function that gets called is event_death, and it comes with a number of parameters:

```
void event_death(object ob, object *killers, object killer, string
  room_mess, string killer_mess) {

  ::event_death (ob, killers, killer, room_mess, killer_mess);
}
```

The first parameter is the object that died. Killers is the array of objects that made up the dead NPC's attacker list. Killer is the object that struck the killing blow, and room_mess and killer_mess are the strings that get displayed to the room and killer respectively.

So, let's make our NPCs a little more responsive to the world around them, and express their disgust if they see someone killed, and their surprise if they were part of the reason they died:

```
void event_death(object ob, object *killers, object killer, string
  room_mess, string killer_mess) {

  if (member_array (this_object(), killers) != -1) {
    init_command ("' Oh, did I do that?", 1);
  }
  else if (ob != this_object()) {
    init_command ("' Oh, how horrid!", 1);
  }

  ::event_death (ob, killers, killer, room_mess, killer_mess);
}
```

You'll see this working if you clone a couple of the girls into your environment, and then set them fighting against each other. After they've exchanged a few hits, you'll see something like this:

```
The upwardly mobile socialite dealt the death blow to the gold digger.
The upwardly mobile socialite asks: Oh, did I do that?
```

If all we want to do is have something special happen when an NPC dies, we don't need to catch the event – we can just provide a second_life function. This gets executed as part of the natural processing of the death code, and can be used like so:

```
int second_life() {
  do_command ("' What a sad ending to a beautiful love story!");
  return 0;
}
```

Now she will express her poignant last words when she is unceremoniously executed by an indifferent creator:

```
The gold digger dies.
The gold digger exclaims: What a sad ending to a beautiful love story!
```

Event handling is one of the most powerful techniques you have available for providing responsive code. The important distinction is that you don't trigger events in your own code, you just provide code that should be executed when the event is triggered externally. The code should sit dormant until the trigger event occurs ‘in the world'.

There are many events that get triggered in the game. For example, we can make our girls even more annoying by having them comment on fights in their environment:

```
void event_fight_in_progress(object me, object him) {
  if (me != this_object() && him != this_object()) {
    if (!random (10)) {
      do_command ("' Stop fighting, you beastly brutes!");
    }
  }

  ::event_fight_in_progress (me, him);
}
```

Each event has its own set of parameters, so you may have to consult the documentation to see exactly what information you are working with. For the event_fight_in_progress method, we have two object parameters - one for the attacker and one for the defender. Thus, we make it so that our NPC only chats if they are not involved in the fight – load_a_chat handles what they should be saying when they are in combat.

Events are not limited to NPCs – we can catch these events in rooms and items too. Imagine for example we wanted our rooms to magically dest any corpse that appears – 'Keep Betterville Tidy' for example. If we want that to happen in a specific room, we can provide an event_enter function. This is called whenever an object enters the environment or the inventory of another object. So we'd override that function in the room where we wanted that to happen:

```
void event_enter(object ob, string mess, object from) {
  if (ob->query_corpse()) {
    ob->move ("/room/rubbish", "$N appear$s with a pop.",
      "$N disappear$s with a pop.");
  }

  ::event_enter (ob, mess, from);
}
```

Kill a socialite in this room, and the following occurs:

```
The upwardly mobile socialite dies.
The corpse of an upwardly mobile socialite disappears with a pop.
```

Be warned – you don't want to die here yourself!

The Power of Inherits

Now, let's see why we went to the trouble of creating inherits for Betterville. One of the things that we can now do is implement area-wide functionality just by putting an event function in an appropriate inherit – for example, if we want to have corpses disappear across the entire village, we put the above event_enter function into the betterville_room inheritable.

Unfortunately when we do this and update all our inherits, we get the following warnings:

```
Updated /d/learning/betterville/inherits/betterville_room.
/d/learning/betterville/inherits/inside_room.c line 36: Warning:
event_enter() inherited from both
/d/learning/betterville/inherits/betterville_room.c and
/std/room/basic_room.c; using the definition in
/d/learning/betterville/inherits/betterville_room.c. before the end of file

Updated /d/learning/betterville/inherits/inside_room.
/d/learning/betterville/inherits/outside_room.c line 37: Warning:
event_enter() inherited from both
/d/learning/betterville/inherits/betterville_room.c and
/std/room/basic_room.c (via /std/room/outside.c); using the definition in /d/
learning/betterville/inherits/betterville_room.c. before the end of file
Updated /d/learning/betterville/inherits/outside_room.
```

Yikes, what does all of that mean??

Well, remember how we talked about the idea of scope resolution? What we have here is one of the problems that multiple inheritance causes. In betterville_room, we have put an event_enter function. However, /std/room/outside already has an event_enter, and /std/room/basic_room also already has an event_enter. These warnings are telling us that the driver can't tell which it should use, and so it has decided to just use one over the other because it doesn't know how to use both.

We resolve this by providing an event_enter in inside_room and outside_room, and have that function handle the execution of the inherited functions. In inside_room:

```
void event_enter(object ob, string mess, object from) {
  basic_room::event_enter (ob, mess, from);
  betterville_room::event_enter (ob, mess, from);
}
```

And then in outside room:

```
void event_enter(object ob, string mess, object from) {
  outside::event_enter (ob, mess, from);
  betterville_room::event_enter (ob, mess, from);
}
```

These functions override the event_enter functionality, and tells the MUD to call the inherited event_enter first in basic_room or outside, and then in betterville_room. This is the usual approach to resolving this kind of errors, but in certain complicated situations more complex, tailored functionality may be required. You can worry about that when it happens to you, though.

Conclusion

Our discussion about NPCs was little more than a chance for us to move the topic on to a more rich vein of inquiry – the power of events in MUD coding. Events are powerful and supported at multiple levels of the MUD – you can associate functionality with a particular thing happening, such as a spell being cast or someone emoting in the room, or one of the many other events that get triggered. That's a lot of power for a small, humble function. You'll have cause to explore the idea further in the code you develop from here on in.

Lady Tempesre

Introduction

We have two additional NPCs that are part of this development – the Beast and Lady Tempesre Stormshadow, the shopkeeper. Part of our creation of these NPCs will be in setting up the relationship between the two – as you will undoubtedly recall from Being A Better Creator, they are former lovers and the Beast still protects her from harm.

There are several things we'll need to put in place to make all of this work – two NPCs for example! We'll also need some way of storing the list of individual's on the Beast's ‘to disembowel' list'.

The Lady

Let's begin with Lady Tempesre Stormshadow – she's the most straightforward of the two. She's designed to be a service NPC, as well as a flavour NPC – she'll set some of the scene of the development.

We've already discussed the way we can trap events to provide situational code triggers – we'll be doing the same thing here. When she is killed, we'll register the name of the player in a ‘disembowel_handler' – she is a service NPC, and so we need to implement some kind of disincentive for players killing her. The rest of the NPC is fairly straight-forward.

Let's start with the core of the NPC:

```
#include "path.h"
inherit INHERITS + "betterville_npc";

void setup() {
  set_name ("stormshadow");
  set_short ("Lady Tempesre Stormshadow");
  add_property ("determinate", "");
  set_long ("This is Lady Tempesre Stormshadow, an older but still "
    "attractive woman with a noble bearing.  Her eyes are bright and "
    "calculating, sizing up the value of all that her gaze falls "
    "upon.\n");
  add_adjective (({"lady", "tempesre"}));
  add_alias (({"lady", "tempesre"}));
  set_gender (2);
  basic_setup ("human", "wizard", 150 + random (50));
  setup_nationality ("/std/nationality/genua", "Genua");

  load_chat(120, ({
    1, "' Come buy one of my beautiful dresses!",
    1, "' Show the Beast you care with a wonderful outfit!",
```

```
    1, "' The Beast wouldn't give his heart to a poorly dressed bumpkin!",
    1, "' Follow your star, straight to my shop!",
    1, "' Beauty is only skin-deep, but you can sort out 'ugly' with a "
       "dress!",
  }));

  load_a_chat(120, ({
    1, "' You don't know how much you're going to regret this!",
    1, "' Can your mother sew?  She'll need to be able to.",
    1, "' Help!  Help!",
    1, "' You're no handsome prince, what do you think you're doing?"
  }));
}
```

Note that we've set her as a wizard – later on, we're going to write some spells for her that fit the theme of the village and make her more of a challenge for those looking to fight her. She doesn't need to be especially clever in terms of the code she has, but she does need to have a range of interesting conversational responses that set the scene. Let's not go overboard here – this is a tutorial, not an actual completed area. But we want lots of responses in a flavour NPC:

```
add_respond_to_with( ({
    "@say",
    ({"where"}),
    ({"you", "tempesre", "stormshadow"}),
    ({"from"}),
  }),
  "' I am Lady Tempesre Stormshadow, hailing from the diamond "
  "city of Genua!");

add_respond_to_with( ({
    "@say",
    ({"about"}),
    ({"genua"}),
  }),
  ":emote shuffles her feet.");

add_respond_to_with( ({
    "@say",
    ({"genua"}),
  }),
  "' Yes, I come from Genua, which is far away from here!  In... uh... "
    "another country!");
```

Our most important response though is that she should link up the context of the development, so she should response to discussion about the beast:

```
add_respond_to_with( ({
    "@say",
    ({"beast"}),
  }),
  "' I knew the beast once.  Once... upon a dream.  Before the curse.");
```

```
add_respond_to_with( ({
    "@say",
    ({"curse"}),
  }),
  "' He was a good man, before Lilith.  Before... I spurned him.");

add_respond_to_with( ({
    "@say",
    ({"lilith"}),
  }),
  "' Lilith!  She of her vicious cruelty!  She cursed him to the form "
    "of a beast as part of her horrible fascination with stories.");

add_respond_to_with( ({
    "@say",
    ({"story", "stories", "cruelty"}),
  }),
  "' Stories are powerful.  I will say no more... but if you should "
    "ever make your way to a library, you could find out what happens "
    "when stories run amok.  Read about Genua.");

add_respond_to_with( ({
    "@say",
    ({"spurned", "spurn", "love", "loved"}),
  }),
  "' I... loved him once.  I still do, in my way.  But I cannot bring "
    "myself to look upon him now.  That is my weakness, my curse.");
```

A good flavour NPC sets the scene and leads to other parts of the development. For example, here we can find out about the reason why the Beast is what he is, his relationship to our NPC, and that ‘Genua' is a topic that may be profitably researched in the library. Clues for the inquisitive can be threaded everywhere, providing a narrative payoff for those who wish to explore as well as showing that some thought has gone into the relationship of all the different elements.

To fit her role as the 'evil witch' of sorts of the development (as discussed in our conception of the village), we should dress her up in some appropriately black and figure-hugging clothes. That's left as an exercise for the reader.

Shopkeeper Disincentives

The nature of our game is to permit player interactions with NPCs even when they have an undesirable impact on the game. We allow players to kill shopkeepers because it is thematically consistent that they be able to do so. However, we disincentive the activity by adding in a punishment when players kill NPCs we'd rather they didn't, or removing any reward associated (by setting the NPC to have no XP reward). Service NPCs are an excellent example of individuals we would like to protect. We even have a thematic way of doing it – our shopkeeper here is to be protected by the Beast. Our question is, how do we do this?

There are several options:

1. When the player attacks Tempesre, a roar goes up in the distance and the Beast makes his way to the shop to protect her. We shall call this the ‘race against time' system.
2. When the player attacks Tempesre, the beast appears instantly and they fight together. We'll call this the ‘two against one' scheme.
3. When the player kills Tempesre, their name gets recorded in a handler, and the beast will attack anyone he sees who is registered with that handler.

Each of these options has its advantages and disadvantages.

1. In the race against time system, of a player is sufficiently high level they will have killed Tempesre before the beast gets there. There is thus no disincentive in place.
2. In the two against one system, it is unrealistic for an NPC to simply pop into existence when needed. For suitably high level players really what we have here is double the XP reward. We could remove the XP reward for both, but that's the easy way out and we learn nothing doing that.
3. With the disembowel handler, if the player never meets the Beast there is no risk of punishment. Additionally, it requires us to code another handler to record transgressions.

For no other reason than it gives us cause to talk about another element of LPC coding, we're going for solution three – the ‘disembowel handler'. We'll talk about 'arranging' visits to the Beast in a later chapter.

The Disembowel Handler

This handler is extremely simple – it needs a way to add players to a list of transgressors, query if players are on that list, and remove them from the list. However, it needs to do something else that our previous handler didn't – it needs to preserve its state over reboots. In other words, it has to save and restore its data.

```
string *to_punish;

void create() {
  seteuid (geteuid());
  to_punish = ({ });
}
```

```
void add_to_punish (string str) {
  if (member_array (str, to_punish) == -1) {
    to_punish += ({ str });
  }
}

int query_to_be_punished (string str) {
  if (member_array (str, to_punish) == -1) {
    return 0;
  }
  return 1;
}
void remove_to_punish (string str) {
  to_punish -= ({ str });
}
```

We also need to include it in our path.h file:

```
#define   INHERITS          BETTERVILLE + "inherits/"
#define   ROOMS             BETTERVILLE + "rooms/"
#define   HANDLERS          BETTERVILLE + "handlers/"
#define   CHARS             BETTERVILLE + "chars/"
#define   PORTRAIT_HANDLER     (HANDLERS + "portrait_handler")
#define   DISEMBOWEL_HANDLER   (HANDLERS + "disembowel_handler")
```

Our handler is simple, but there's no reason it has to be shoddy. There are two kinds of methods that are used in handlers – the first set of methods define the ‘interface' of the handler – they're the set of methods that developers should be using in their own code to interact with it. The rest of these are internal methods – methods that exist to make the work of the handler progress smoothly. Often these will be set as private methods so that no-one can accidentally make use of them but it can be useful as a developer for the handler coder to be able to query them directly with calls and execs. That can't be done with private methods.

The external interface methods should make sure they are not easily flummoxed by other coders making use of the calls – for example, what happens if they pass the reference of a player rather than the name of a player? That kind of thing is easy to compensate for in our code, and so we should:

```
private string query_player_name (mixed player) {
  if (objectp (player)) {
    return player->query_name();
  }
  if (stringp (player)) {
    return player;
  }
  return "Er, what?";
}
```

This method will take in a mixed parameter – if it's an object it gets, it returns the query_name of the object. If it's a string, it returns the string itself. Otherwise, it returns a confused query. We can then build that into each of our methods:

```
string *to_punish;

void create() {
  seteuid (geteuid());
  to_punish = ({ });
}

string query_player_name (mixed player) {
  if (objectp (player)) {
    return player->query_name();
  }
  if (stringp (player)) {
    return player;
  }
  return "Er, what?";
}

void add_to_punish (mixed player) {
  string str = query_player_name (player);

  if (member_array (str, to_punish) == -1) {
    to_punish += ({ str });
  }
}

int query_to_be_punished (mixed player) {
  string str = query_player_name (player);

  if (member_array (str, to_punish) == -1) {
    return 0;
  }
  return 1;
}

void remove_to_punish (mixed player) {
  string str = query_player_name (player);
  to_punish -= ({ str });
}
```

Now all we have to do is bind her death into the appropriate handler call:

```
int second_life() {
  object *enemies = this_object()->query_attacker_list();

  do_command ("' The Beast... shall hear of this...");

  foreach (object enemy in enemies) {
    DISEMBOWEL_HANDLER->add_to_punish (enemy);
  }
}
```

The last piece of this puzzle will be to have the Beast himself query if the people he meets are due for punishment, but we'll come to that in the next chapter.

Data Persistence

The last thing we need to discuss is how to make our handler save its state – it shouldn't be the case that people can wait out a reboot and then visit the Beast in safety. If something is designed as a disincentive, it should produce a genuine reason not to do it.

There are several ways in which data persistence is handled in the MUD. We'll talk about the simplest of these in this chapter. First of all, let's talk about how the MUD represents data in a file.

Every data type has a particular kind of text representation that the driver knows how to parse into an variable reference. When an object is saved in its entirety, it is saved as a .o file, and that .o file contains the state of each variable in the object, except those marked as nosave, like so:

```
nosave int do_not_save_this;
```

When an object is saved, the state of all its variables are recorded and written to a file. For example, the .o file associated with my bank account is as follows:

```
#/obj/handlers/bank_handler.c
accounts (["Genua National":13641051,"Bing's First":30000,
"LFC":41548140,"test":330000,"Bing's Bank":16000,"Klatchian
Continental":14,])
```

This file tells the MUD that the save file was generated by the bank handler, and it consists of an ‘accounts' variable containing the listed entries. When the MUD reloads this file, it knows to create the following mapping:

```
([
  "Genua National":13641051,
  "Bing's First":30000,
  "LFC":41548140,
  "test":330000,
  "Bing's Bank":16000,
  "Klatchian Continental":14,
])
```

It also knows that this mapping should be restored to the handler as the variable set as ‘accounts'.

We tell the MUD to save an object by using the **save_object** efun. However, this sometimes needs a little more instruction to the MUD because of the way our permissions system works.

Effective User Identifiers

Every object on the MUD has what is called an **euid**, which stands for ‘Effective User Identifier'. For creators, your euid is the same as your username. For players, the euid is PLAYER. For rooms, NPCs and such, it depends on in which directory the object is located. For domain directories, the euid will be the same as the domain in which the object resides (so for /d/learning/, the euid is ‘Learning'. Other objects have a default euid according to the following table:

Directory	Default EUID
/secure/	Root
/obj/	Room
/std/	Room
/cmds/	Room
/soul/	Room
/open/	Room
/net/	Network
/www/secure/	Root
/www/	WWW

The euid defines what access an object has to read and write from other directories on the MUD. Objects in a /w/ directory have the same permissions as the person who owns the directory – so if the creator *draktest* has write to /save/, so too do all objects in /w/draktest/.

To see what directories each of the above euids has access to, you can use the following command:

```
permit summary <euid>
```

For example:

```
> permit summary WWW
Euid        Path
WWW     R   /save/deities
WWW     R   /save/wizard_orders
WWW     RW  /save/www
WWW     R   /secure
WWW      W  /www/external/includes
WWW      W  /www/new/includes
WWW     RW  /www/osrics/results
```

The consequence of this is that if an euid doesn't have write access to a directory, it can't write there. This ensures at least some measure of protection against rogue objects doing Bad Things.

Now, this may seem like a fairly irrelevant distraction, but it relates to how saving and restoring objects works in the MUD. Let's add a function that saves the status of the handler. First, we add a new define:

```
#define   SAVE            BETTERVILLE + "save/"
```

And then we add the function to the disembowel_handler:

```
void save_me() {
  save_object (SAVE + "disembowel_save");
}
```

Now, when we call this function in our own directory, it works absolutely fine. However, if someone else calls this function in our directory (someone who does not have write access to the save directory we defined) it'll give the following message:

```
Denied write permission in save_object().
```

This is because when an interactive object (such as a player or creator) is the source of a call like this, the MUD notes it and says ‘Oh-ho, you're not permitted to do this' and gives an error even though the directory in which the object resides has access. To get around this, we must say to the MUD ‘it's okay, you can trust this guy. It's cool', which you do by using the **unguarded** function:

```
void save_me() {
  unguarded ((: save_object (SAVE + "disembowel_save") :));
}
```

Unguarded says to the MUD ‘this is far as you need to have checked. If this object has permission, then whoever causes this function to trigger also has permission'. It's safe to do this for pretty much any code that goes into the game – anyone who knows how to use this kind of thing For Evil already has access to worse ways to break our security model.

For restoring the state of the handler, we use the restore_object function:

```
void load_me() {
  unguarded ((: restore_object (SAVE + "disembowel_save") :));
}
```

All that is left at this point is to tie these two functions into the rest of the handler – load the handler when the object is created, and save the object whenever its internal state changes;

```
#include "path.h"

string *to_punish;

void load_me();
void save_me();

void create() {
  seteuid (geteuid());
  to_punish = ({ });
  load_me();
}

string query_player_name (mixed player) {
  if (objectp (player)) {
    return player->query_name();
  }
  if (stringp (player)) {
    return player;
  }
  return "Er, what?";
}

void add_to_punish (mixed player) {
  string str = query_player_name (player);

  if (member_array (str, to_punish) == -1) {
    to_punish += ({ str });
    save_me();
  }
}

int query_to_be_punished (mixed player) {
  string str = query_player_name (player);
  if (member_array (str, to_punish) == -1) {
    return 0;
  }
  return 1;
}

void remove_to_punish (mixed player) {
  string str = query_player_name (player);
  to_punish -= ({ str });
  save_me();
}

void save_me() {
  unguarded ((: save_object (SAVE + "disembowel_save") :));
}

void load_me() {
  unguarded ((: restore_object (SAVE + "disembowel_save") :));
}
```

One last thing to discuss about saving objects – we can also tell the MUD to compress save files. Sometimes this is a good thing to do, especially when working with large amounts of data – we just pass an additional parameter to save_object, like so:

```
void save_me() {
  unguarded ((: save_object (SAVE + "disembowel_save", 2) :));
}
```

Now when the handler saves it saves as a gzipped file – for big files, this will save huge amounts of disk space and file I/O at a cost of a little CPU. It's a bargain whatever way you look at it!

Conclusion

Our NPC in this chapter is not an especially complicated one, but it led us into a discussion of the way file permissions work on the MUD, and what we can do to restore the state of objects when we need it. Data persistence is not tricky to do – there's not much more to it than we have talked about here. For very complex, or very intensive file I/O, we may need to do something a little more tailored. For most objects and handlers, this is as much as you ever need to know.

Beastly Behaviour

Introduction

Now that we've happily implemented our dozy girls and our shopkeeper, let's write up the last of our NPCs. The Beast is a complicated character, but we're not going to have him as a boss NPC or even an especially unique combat opponent. He's just going to rant and rave and be a narrative reward for those who make it to the top of our ruined library.

We already know the tools for most of this, although we'll also make the beast throw a few unique attacks out there too, for the sake of variety.

```
#include "path.h"
inherit INHERITS + "betterville_npc";

void setup() {
  set_name ("beast");
  set_short ("The Beast");
  add_adjective ("the");
  add_property ("determinate", "");

  set_long ("That the Beast was once man would never be apparent from his "
    "appearance.  The magic that worked through him has shed away every "
    "last trace of humanity, leaving only a core of pure animal rage.\n");
  add_adjective (({"lady", "tempesre"}));
  add_alias (({"lady", "tempesre"}));

  set_gender (1);

  basic_setup ("werewolf", "warrior", 400 + random (50));

  setup_nationality ("/std/nationality/genua", "Genua");

  load_chat(120, ({
    1, ": sits up on his haunches and mournfully grooms himself.",
    1, "@sigh sadly!",
    1, ": stares sadly into space",
  }));

  load_a_chat(120, ({
    1, "#animal_growling",
    1, "@roar",
  }));
}
```

We want his growling to be vaguely realistic, and animals don't growl the same way each time – so let's write a function to do some semi-randomised growling, like so:

```
void animal_growling() {
  string *start_letters = ({"Gra", "Ra", "Kra", "Bwa"});
  string *start_bit = allocate (3 + random (4), "a");

  string *mid_bit = allocate (3 + random (4), "r");
  string *end_bit = allocate (3 + random (4), "a");

  do_command ("say " + element_of (start_letters)
    + implode (start_bit, "") + implode (mid_bit, "")
    + implode (end_bit, "") + "r!");
}
```

Implode is a function that takes an array and condenses it down to a string, and allocate lets us create an array – the first parameter is how many elements the array will have, and the second is the starting value to give each element of the array. The result is that the Beast growls vaguely realistically:

```
The Beast exclaims: Graaarrraar!
The Beast exclaims: Raaaarrraaar!
The Beast exclaims: Graaarraaaar!
The Beast exclaims: Bwaaaarraar!
```

There's no real need for that of course, it's just a l'il bit of sugar for our star attraction.

In Being A Better Creator, we talked about the Beast being damaged by his experience and unable to express himself properly. We can handle this by giving each response an unmangled form, and pass it through a mangling function. Our mangling function will change random parts of the string into grunts, groans and moans. We should also make sure that we reuse code where we can:

```
string random_growling_bits() {
  string *start_bit = allocate (3 + random (4), "a");
  string *mid_bit = allocate (3 + random (4), "r");
  string *end_bit = allocate (3 + random (4), "a");

  return implode (start_bit, "") + implode (mid_bit, "") +
    implode (end_bit, "");
}

void animal_growling() {
  string *start_letters = ({"Gra", "Ra", "Kra", "Bwa"});

  do_command ("say " + element_of (start_letters) + random_growling_bits()
    + "r!");
}

string contort_word (string word) {
  string start = word[0];
  string end = word[<1];
  return start + random_growling_bits() + end;
```

```
}

string mangle_string (string str) {
  string *arr = explode (str, " ");
  for (int i = 0; i < sizeof (arr); i++) {
    if (!random (3)) {
      if (sizeof (arr[i])) {
        arr[i] = contort_word (arr[i]);
      }
    }
  }
  return implode (arr, " ");
}
```

It's the mangle_string function here that handles changing a chat into a contorted string – at random intervals throughout the string, it'll use the contort_word method to turn a word like ‘Hello' into the word 'haaarraarrro' or such. Try to update this NPC though, and we get the following:

```
Type mismatch ( string vs int ) when initializing end before the end of line
```

This is all down to how the MUD actually represents individual characters of a string – it stores them as numeric representations of the letter they are supposed to be. Technically they get stored as ASCII codes, each number representing a particular letter. So when we get a single character off of an string (such as word[0]), what we get out is the number representing that letter. As such we need to store them in ints:

```
string contort_word (string word) {
  int start = word[0];
  int end = word[<1];

  return start + random_growling_bits() + end;
}
```

Now it compiles, but we have a new problem. Try to call the function with a testing string, and we get something rather strange back:

```
call mangle_string ("hello world hello world hello world") beast
Returned: "hello world 104aaaaarrraaaa111 119aaaarrrrrraaaaaa100 hello world"
```

Say what? What? Why is this!

Well, I'm sure you can guess – we represented letters coming off of our string as integers, and so that's how they get added to the string - as numbers. If we want to turn these numbers into characters, we need to pass them through sprintf and tell it ‘treat this number I give you as a character' – we do that with the %c code:

```
string contort_word (string word) {
  int start = word[0];
  int end = word[<1];

  return sprintf ("%c", start) + random_growling_bits()
    + sprintf ("%c", end);
}
```

To see what's actually happening here, try this little exec:

```
exec int num = 97; for (int i =0; i < 26; i++) { printf ("%c\n", num + i); }
```

This will print out all the letters of the alphabet. To see it print out the numbers instead, use the %d code for sprintf:

```
exec int num = 97; for (int i =0; i < 26; i++) { printf ("%d\n", num + i); }
```

This second exec does the same thing as the first, except what you'll get out of it are the numeric ASCII codes.

Some C-type languages have a ‘char' datatype to simplify the process. Unfortunately LPC doesn't, and so we need to make use of these kinds of work-arounds.

Anyway, now we bind that into his add_response_to_with to provide tailored, yet randomized responses:

```
add_respond_to_with( ({
    "@say",
    ({"lilith"}),
  }),
  "#lilith_response");
```

And then the function that handles giving the random output:

```
void lilith_response() {
  string text = "Her!  A thousand curses upon her and her kin!  May "
    "the flesh be rent from her bones for her cruelty!  It was she who "
    "cursed me to this shape!";

  init_command ("say " + mangle_string (text), 1);
}
```

Ask him about Lilith and taste his pain!

```
The Beast exclaims: Her! A taaaarrraaad curses upon her aaaaaarrrrrraaad
haaarrraaaar kin! May the flesh be rent from her baaaarrrrraaaaaas for
haaaaaarrrrrraaar cruelty! It was she waaaaarrraaao cursed maaaaaarrrraaaae
taaarrrrraaaaaao this saaaaarrraaaaaa!
```

We can add as many of these responses as we want, and the story will unfold with successive telling of the tale. We can add new responses quickly and easily now we have the framework in place. One for saying the word ‘beast' for example:

```
add_respond_to_with( ({
    "@say",
    ({"beast"}),
  }),
  "#beast_response");
```

And the function to handle it:

```
void beast_response() {
  string text = "Oh, fairy-tales are in the lifeblood of Genua.  Our land "
   "is forever changed by the power of raw and untamed narrative.  I am one "
   "of the victims of that legacy.";

  init_command ("say " + mangle_string (text), 1);
}
```

We can be as detailed as we like here, and if we want the horrible grunts and growls to be even more effective, we just need to change the mangle_string function to make it so.

Combat Actions

Now that we've got a framework for story-telling in the beast, let's make him a little more interesting to fight by giving him some unique combat actions.

Combat actions are added using the add_combat_action function, which takes three parameters - we saw this in LPC For Dummies 1, but we didn't really emphasise it. The first parameter is the ‘chance' that the attack will occur. The second is the name of the attack, which is an identifier for us if we need to refer to it later. The third parameter is the action to occur. This is usually a string (which gets executed directly), a function pointer, or an array. If the array has one element, the function named gets called on the object in which the action is defined. If it's two elements, the function gets called on the specified object.

So, let's try one of these for the beast. Let's give him the shove command first. We do this using add_known_command in setup:

```
add_known_command ("shove");
```

And then we can add a combat action (again in setup)

```
add_combat_action (25, "shove people", ({"shove_enemy"}));
```

The array indicates that the method ‘shove_enemy' will get called on the beast whenever the MUD decides to execute this combat action. Alas, setting the chance of combat actions is an inexact science at best, and you will need to experiment until you find a value that appeals. Higher means the attack will be executed more often. Essentially it's the chance per combat round that an action is called, so our shove above will occur, on average, once every four rounds.

```
void shove_enemy (object attacker, object defender) {
  object target;

  if (attacker == this_object()) {
    target = defender;
  }
  else {
    target = attacker;
  }

  do_command ("shove " + file_name (target));
}
```

The nice thing about combat actions is that they don't need to be linked to actual game commands, although it's nice if they are. We could for example add a little bit of healing to the beast:

```
add_combat_action (25, "lick_wounds", ({"lick_wounds"}));
```

And then the function to go with this:

```
void lick_wounds (object attacker, object defender) {
  if (query_hp() < 500) {
    init_command (": licks at his wounds for a moment, and seems to "
      "be healed a little from the process.", 1);
    adjust_hp (200);
  }
}
```

It's better on the whole if we use existing commands and spells, because these are likely more balanced than the numbers we'll just slot into creatures. It's an option you have available though where appropriate.

Smart Fighters

You'll have noticed that many NPCs across the game make use of fairly solid (albeit unimaginative) tactics as they fight with players. They launch specials, trip people up, use the right kinds of weapons and use the right kind of defences. We have an inherit that handles all of that called the smart_fighter inherit. If we want our beast to be a bit more clever, we can make use of this.

However, we need to either use /obj/monster/smart_fighter_monster for our base inherit, or build our own custom inherit to go along with our current NPC inherit. We can't use the former, because we want all NPCs to spawn from a common inherit so we can make any changes we want to. What we need to do is change our NPC inherit so that we have the choice of using smart fighter if we like. We can do that just by changing the inherit being used:

```
inherit "/obj/monster/smart_fighter_monster";

void create(){
  do_setup++;
  ::create();
  do_setup--;

  if ( !do_setup ){
    this_object()->setup();
    this_object()->reset();
  }
}
```

Now, when we want to setup an NPC to behave properly, we use the set_smart_fighter function in setup. We also need to #include smart_fighter.h to make sure we have access to all the necessary defines for this.

Our beast is unarmed, so we want him to use unarmed attacks. We also want him to swap between dodge and parry because he doesn't have any particular preference for either. As such, our set_smart_fighter function call will look like this:

```
set_smart_fighter (USE_UNARMED, DEFEND_BALANCED);
```

The smart fighter code does a lot of things, such as making sure that NPCs have sensible amounts of necessary skills and that they have access to the necessary commands. Most importantly though, the smart fighter code lets you set some moderately sophisticated behaviour in your NPC by passing in the right values to the set_smart_fighter functions. In fact, we no longer need our function for shoving people – smart fighter will do that for us. Thus, away it goes.

We can provide a third parameter – one that handles how our NPC reacts to spell-casting. If we want to be able to have an NPC that runs away from spells, we can do that:

```
set_smart_fighter (USE_UNARMED, DEFEND_BALANCED, SPELL_REACT_RUN_AWAY);
```

Now, when we cast an offensive spell, we see the following behaviour:

```
You prepare to cast Pragi's Fiery Gaze.
You lazily close your hand around the eye.
The Beast leaves northeast.
```

Our Beast is thus Unfriendly to Spell-casters, but that's not really the point. The point is – we achieve this effect by using the right parameters to our set_smart_fighter call, and nothing else.

Bitwise Operators

We can also define combinations of actions with this function, albeit by using a new and unfamiliar syntax. If we want our NPC to use a balance of sharp and pierce, then we'd use the following as the first parameter to set_smart_fighter:

```
(USE_SHARP | USE_PIERCE)
```

The symbol here between the two is called a **bitwise operator**, specifically a **bitwise or**. Everything in your computer is, at the base, represented as a binary number – a number made up of 1s and 0s. Where we have decimal numbers, they work according to a base 10 system – we naturally work in multiples of 10. Binary is a base two system – it's constructed of multiples of two. A decimal number then:

100	10	1
1	2	3

Each column is ten times the one before. This number is the familiar 123 gained by the following simple arithmetic:

```
(100 * 1) + (10 *2) + (1 * 3)
```

Binary works exactly the same way except each column is two times the one before:

128	64	32	16	8	4	2	1
0	*0*	*0*	*1*	*0*	*1*	*0*	*1*

The number 00010101 is represented by the following arithmetic:

```
(0 * 128) + (0*64) + (0 * 32) + (1 * 16) + (0 * 8) + (4 * 1) + (2 * 0) + (1 * 1)
```

The number then is 21 in decimal. The specifics of this you can practice on your own time, it's not especially important beyond the fact that numbers can be represented as ones and zeros.

Bitwise operators allow you to do the familiar AND and OR operators on each bit in a binary number. We can use a single bar for a bitwise or, and a single ampersand for a **bitwise and**. For example, if we did 22 & 10, the number we ended up with would be the result of an AND operation on each bit of the number – only if both bits are 1 will it make it into the final number:

	16	8	4	2	1
22 in Binary	1	0	1	1	*0*
10 in Binary	1	0	0	1	*0*
Result of AND	*0*	*0*	*0*	*1*	*0*

The result of a bitwise and on 22 and 10 is thus the number 2:

```
exec return 22&10  Returns: 2
```

A bitwise Or works the same way, except that the bit will end up in the number if either of the provided numbers have a 1 in that column:

	16	8	4	2	1
22 in Binary	1	0	1	1	*0*
10 in Binary	1	0	0	1	*0*
Result of OR	*1*	*1*	*1*	*1*	*0*

The result of a bitwise or on 22 and 10 is thus the number 30:

```
exec return 22|10  Returns: 30
```

It is this process at work inside the smart fighter code – each of the defines is a binary number, and by using these operators on them we can tell the code what blend of options we require. From smart_fighter.h:

```
#define USE_SHARP 0x0008  #define USE_PIERCE 0x0004
```

This a new notation for numbers, and it simply tells LPC that we are defining a hexadecimal (base 16) number rather than a decimal number. Don't worry about why this is so, the principles are exactly the same - hexadecimal maps precisely onto binary.

When we Bitwise OR these together, we end up with the number 12. Every combination of possible values has a unique result for this bitwise or comparison, and we can later use the bitwise and operator to see whether or not a specific value is represented. For example, let's use (USE_SHARP | USE_PIERCE), which breaks down into (8 | 4) which in itself is:

	16	8	4	2	1
8 in Binary	0	1	0	0	*0*
4 in Binary	0	0	1	0	*0*
Result of OR	*0*	*1*	*1*	*0*	*0*

Later on, if we want to know whether or not someone has set USE_SHARP, we would do the following:

	16	8	4	2	1
Fighter settings	0	1	1	0	*0*
USE_SHARP	0	1	0	0	*0*
Result of AND	*0*	*1*	*0*	*0*	*0*

Voila, since we get USE_SHARP as a result out of the bitwise and we know that we should treat this NPC as if it were a sharp user. Likewise, if we wanted to check if it used pierce:

	16	8	4	2	1

Fighter settings	0	1	1	0	*0*
USE_PIERCE	0	0	1	0	*0*
Result of AND	*0*	*0*	*1*	*0*	*0*

Voila, the result of the AND indicates that we should indeed use pierce too for this NPC.

All of this is slightly arcane and you don't *really* need to know how it all works. It is enough to know that it does work, but you'll find this kind of system to be used in several places throughout the MUD, and a sound understanding of Why It Is So will serve you well.

Relating Back to Smart Fighter

Now, why did I take you on that magical mystery tour? It's because of what we can actually do with smart fighter by using bitwise operators. Some of the options you have available are:

Flag	Arg #	Description
USE_COVERT	1	Hide, Backstab, abscond, etc
USE_PIERCE	1	Use piercing weapons and specials
USE_SHARP	1	Use sharp weapons and specials
USE_BLUNT	1	Use blunt weapons and specials
USE_UNARMED	1	Use unarmed weapons and specials
USE_BALANCED	1	A combination of pierce, sharp, blunt and unarmed
DEFEND_DODGE	2	Defend using dodge
DEFEND_PARRY	2	Defend using parry
DEFEND_BALANCED	2	Defend using dodge and parry
SPELL_REACT_ATTACK	3	Attack if someone starts casting a hostile spell
RITUAL_REACT_ATTACK	3	Attack if someone starts casting a hostile ritual
SPELL_REACT_RUN_AWAY	3	Run away if someone casts a spell
RITUAL_REACT_RUN_AWAY	3	Run away if someone performs a ritual

HELP_SAME_NPCS	3	Join in combat if an NPC with the same base_name is under attack
JOIN_ANY_FIGHT	3	Jump into combat if any combat is ongoing in the room

So for example, if we wanted a dagger-wielding coverty NPC that uses dodge, attacks when hostile spells are cast and runs away when rituals are cast, we'd do the following:

```
set_smart_fighter ( (USE_COVERT | USE_PIERCE), DEFEND_DODGE,
(SPELL_REACT_ATTACK | RITUAL_REACT_RUN_AWAY));
```

Pretty cool, eh? All that power, so little effort. It's enough to make you touch your nose with excitement.

Bitwise and Arrays

One last thing for this chapter, and then we'll stop. The bitwise operators we have discussed have an additional usage – they can be used on arrays to perform the same functions as set theory. That's tremendously powerful and useful. For example, imagine the following:

```
string *arr1 = ({"a", "b", "c"});
string *arr2 = ({"a", "c", "d"});
```

If we wanted a quick way to get those elements that exist in both arrays, we can do the following:

```
string *arr3 = arr1 & arr2;
```

The variable arr3 will then contain the array ("a", "c"});

If we want elements that exist in one array or the other, then we can do:

```
string *arr3 = arr1 | arr2;
```

This would give us the array ({"a", "b", "c", "d"}).

Alas, we have no way of doing a bitwise operation that gets elements that are in one array **or** the other, but we never know what lovely things the next driver updates will bring us. There is an exclusive or (XOR) bitwise operator (^), but at the time of writing it does not work on arrays.

Conclusion

We've gone through some fairly intense material in this chapter – bitwise operators are pretty straightforward, but only if you're already comfortable with thinking in binary. However, as time goes by you'll find that easier and easier to do. Even if you only ever use bitwise operators on arrays, they still give remarkable control for very little code, and you will benefit immensely from becoming conversant with them.

The Beast is now complete, such as he ever will be, and with him we end the bulk of the development of our three sets of NPCs. We'll return in the next few chapters to Lady Tempesre to give her some unique spells, but we are otherwise done. We've made a lot of progress in the past few chapters – we've learned about events, data persistence, bitwise operators and the smart fighter code. Combining all of these things together allows us to create extremely flexible and reactive NPCs. Exciting times await those players who fall foul of our plans!

Shopping Around

Introduction

There's a feature we haven't yet addressed in Betterville – that of our shop catering to the whims of the dozy girls. That's a major deficiency as things stand at the moment, because the Dozy Girls should all be clad in the garments that the shop sells. At the moment, all our women are walking around naked, and that is shocking and upsetting. Or shocking at least.

In this chapter we are going to revisit the idea of shops and items, and look at some of the advanced topics that allow us to configure how they look and behave, as well as how we can provide a framework for storing information about items between logins.

Item Shop Inherit

First of all we need an inherit – everything in Betterville is coming from a common set of inherits to give us the greatest degree of flexibility over our code. Unfortunately, we do not yet have one for an item shop and we are going to need one of those. Luckily, it's no great hassle – basic inherits are not difficult for us to provide by now. However, we can't simply just create an item shop inherit – we already have functionality that we want to represent in every room, shops included. Our item shop inherit thus needs us to inherit betterville_inside and also the item shop inherit.

```
#include "path.h"
inherit INHERITS + "inside_room";
inherit "/std/shops/inherit/item_shop";

void create(){
  do_setup++;
  inside_room::create();
  item_shop::create();
  do_setup--;

  set_object_domain ("learning");
  add_property ("place", "Genua");

  if ( !do_setup ){
    this_object()->setup();
    this_object()->reset();
  }
}
```

This causes our familiar error messages about methods being inherited from two separate objects, and so we need to provide over-ride methods for dest_me, event_theft, and init. These three methods are the ones that appear in the warnings when we update our inherit 'as is'. The dest_me and init methods are straightforward:

```
void dest_me() {
  inside_room::dest_me();
  item_shop::dest_me();
}

void init() {
  inside_room::init();
  item_shop::init();
}
```

The event_theft however requires us to deal with parameters (because when the event is triggered, the triggering code sends in a pile of parameters to make things work smoothly). As such, that method looks a little different:

```
void event_theft(object command_ob, object thief, object victim,
  object *stolen ) {

  inside_room::event_theft (command_ob, thief, victim, stolen);
  item_shop::event_theft (command_ob, thief, victim, stolen);
}
```

Provide these, and we're ready to roll – we have our very own item shop inherit that we built with our own two little hands.

An Item Shop

Now, let's make something of this little inherit we put together – we'll create the first steps of our own item shop in the village square. We setup the basic room and add the code to deal with our shopkeeper NPC:

```
#include "path.h"
inherit INHERITS + "betterville_item_shop";

object lady;

void setup() {
  set_short ("Lady Tempesre's Fine Garments");
  add_property ("determinate", "");
  set_long ("This is a shop full of beautiful clothes for beautiful "
    "girls.  The garments are each fit for a princess, with price-tags "
    "to match.\n");
  set_light (100);
```

```
  add_exit ("south", ROOMS + "betterville_06", "door");
}

void reset() {
  ::reset();
  call_out ("after_reset", 3);
}

void after_reset() {
  lady = load_object (CHARS + "lady_stormshadow");
  if (environment (lady) != this_object()) {
    lady->move (this_object(), "$N appear$s from behind a shelf.",
      "$N run$s off to $p shop.");
  }
}
```

This is all fairly straightforward, and it's content we covered in LPC For Dummies 1. However, what it gives us is a base for what we want to do now – create a range of beautiful garments!

Item Development

We spoke about clothes and virtual files in the first LPC for Dummies, so we won't rehash that here. First of all, let's write a basic dress for the shop. We won't write it as a virtual file (for reasons that will become clear later), we'll do it as a normal LPC .c file. First of all though, we need to talk about where we're going to save these files.

The logical place to save them, considering the way we have been saving things so far, is to save them in an ‘items' sub-directory of Betterville. Unfortunately, this doesn't work as well as we might hope – the armoury will not pick items out of sub-directories in this way. The only way the armoury knows where to find objects is if they are stored in /d/some_domain/items or a sub-directory of same. This is a huge drawback for the way we've been doing things – it means we suddenly need to switch from storing things in one easily explored directory and navigate instead across two different directories.

We're not going to do that – we are going to be awkward and store things in betterville/items. This is purely something we are doing to make these tutorials hang together properly – when developing code in the ‘real world', it should always be accessible from the armoury. Let this be a warning!

Anyway, we're doing the following:

```
#define   ITEMS           BETTERVILLE + "items/"
```

And then we're going to create a simple dress item in that directory:

```
inherit "/obj/clothing";
void setup() {
  set_name ("dress");
  set_short ("beautiful blue dress");
  add_adjective (({"beautiful", "blue"}));
  set_main_plural ("beautiful blue dresses");
  set_long ("This is a beautiful dress, fit for a princess.  It is made "
    "from blue silk and billows around the wearer as if they had some "
    "kind of heating grate at their feet.\n");
  set_weight (8);
  set_value (30000);
  setup_clothing (2000);
  set_damage_chance (30);
  set_material ("silk");
  set_colour ("blue");
  set_type ("long dress");
}
```

And then we are going to add that dress to the shop, in the Usual Manner (in the setup of our clothes shop):

```
add_object (ITEMS + "beautiful_dress", 3 + random (3));
```

So far, so straightforward. Now, let's get a little more adventurous.

Auto Loading

When a player saves, we go through a process whereby we take their skills, tell history, health, guild points, and all the other things that make up that unique player and we store that info in a file on disk. As part of that process, we also store their inventory, including all those elements of each item that are subject to change as the game goes on, such as enchantment levels, condition, engravings, and so forth. This is a fairly complicated procedure, and is known as the **auto load** system. The consequence of this is that creating a player item that saves its state doesn't work like creating a handler that saves its state – we need to hook into the auto load process.

Each object that inherits /std/object has a method called ‘query_dynamic_auto_load' defined in it. It is this function that contains all of the information that makes one object unique and distinct from another object. For example, if I call it on a scabbard in my inventory, I get the following output:

```
Returned: ([ /* sizeof() == 5 */
  "dynamic" : 0,
  "length" : 38,
  "width" : 3,
  "::" : ([ /* sizeof() == 2 */
    "::" : ([ /* sizeof() == 10 */
```

```
      "enchantment time" : 1221841818,
      "properties" : ([ /* sizeof() == 2 */
        "virtual clone" : "/obj/scabbard",
        "shop type" : "armoury",
       ]),
      "enchantment" : 0,
      "degrade enchantment" : 3,
      "keep" : 0,
      "materials" : ({ /* sizeof() == 1 */
        "leather"
      }),
      "cloned by" : "drakkos",
      "read mess" : ({ }),
      "identify" : 0,
      "light" : 0,
    ]),
    "inv" : ({ }),
  ]),
  "wear" : ([ /* sizeof() == 4 */
    "worn" : 1,
    "immune" : ({ /* sizeof() == 4 */
      "cold",
      "sound",
      "gas",
      "mental"
    }),
    "wear effects" : ({ }),
    "condition" : ([ /* sizeof() == 2 */
      "cond" : 9800,
      "lowest cond" : 7350,
    ]),
  ]),
])
```

In there is all the information the MUD needs to take a basic scabbard and turn it into the scabbard I know I love. This mapping gets built through successive specializations of the method in each object that inherits another. Essentially each file takes the responsibility for adding its own state to this mapping, and then passing the call along the line to the next inherit in the chain. Let's look at query_dynamic_auto_load as defined in the scabbard:

```
mapping query_dynamic_auto_load() {
  return ([
    "::" : container::query_dynamic_auto_load(),
    "wear" : wearable::query_dynamic_auto_load(),
    "length" : this_object()->query_length(),
    "width" : this_object()->query_width(),
    "dynamic" : _dynamic
  ]);
}
```

Notice that as part of this method, it makes a call to the query_dynamic_auto_load of its two parent classes, container and wearable:

```
"::" : container::query_dynamic_auto_load(),
"wear" : wearable::query_dynamic_auto_load(),
```

The first line here takes whatever comes out of query_dynamic_auto_load for the container, and stores it in the mapping key "::", and whatever comes out of the wearable call gets stored in the "wear" key. These values for these keys thus contain the mappings that come out of the parents, and the parents themselves contain mappings that come out of their parents:

```
  "::" : ([ /* sizeof() == 2 */
    "::" : ([ /* sizeof() == 10 */
      "enchantment time" : 1221841818,
      "properties" : ([ /* sizeof() == 2 */
        "virtual clone" : "/obj/scabbard",
        "shop type" : "armoury",
       ]),
      "enchantment" : 0,
      "degrade enchantment" : 3,
      "keep" : 0,
      "materials" : ({ /* sizeof() == 1 */
        "leather"
      }),
      "cloned by" : "drakkos",
      "read mess" : ({ }),
      "identify" : 0,
      "light" : 0,
    ]),
    "inv" : ({ }),
  ]),
```

Let's take a short break from this, and go back to our dress...

Dressing Up

Let's say I want these dresses to be a little more configurable than normal dresses. In addition to having the short and the long, I want them to have a little motif I can set. I don't want to have to create a dozen versions of this same dress just so I can have a motif, so I add a variable to the dress:

```
inherit "/obj/clothing";
string motif;

void setup() {
  set_name ("dress");
  set_short ("beautiful blue dress");
  add_adjective (({"beautiful", "blue"}));
  set_main_plural ("beautiful blue dresses");
  set_long ("This is a beautiful dress, fit for a princess.  It is made "
    "from blue silk and billows around the wearer as if they had some "
    "kind of heating grate at their feet.\n");
  set_weight (8);
  set_value (30000);
  setup_clothing (2000);
  set_damage_chance (30);
  set_material ("silk");
  set_colour ("blue");
  set_type ("long dress");
  add_extra_look (this_object());
}

void set_motif (string m) {
  motif = m;
}

string query_motif() {
  return motif;
}

string extra_look (object ob) {
  if (query_motif()) {
    return "The dress has a motif of " + query_motif() + " stitched "
      "onto it.\n";
  }
  return "";
}
```

Now when I clone one and call set_motif, I get the following:

```
This is a beautiful dress, fit for a princess.  It is made from blue silk and
billows around the wearer as if they had some kind of heating grate at
their feet.  The dress has a motif of dancing monkeys stitched onto it.  It
is in excellent condition.
```

Lovely, just what I have always wanted. Alas, it is ephemeral, like the love of a young girl. If we log off and then back on, then the motif disappears:

```
This is a beautiful dress, fit for a princess.  It is made from blue silk and
billows around the wearer as if they had some kind of heating grate at
their feet.  It is in excellent condition.
```

We know we wanted the motif to be saved. The MUD alas does not. This is where we need to hook into the auto load code.

Back To The Auto Loading

See, this auto load stuff is not a magical mystery tour – this kind of state storing is a very common thing creators want to do. It's not actually hard for us to hook into this process, but it's useful to understand why we're doing it rather than assuming it to be some kind of magical transaction. We need to provide our own specialization of query_dynamic_auto_load to store the motif along with everything else:

```
mapping query_dynamic_auto_load() {
  return ([
    "::" : ::query_dynamic_auto_load(),
    "motif" : query_motif(),
  ]);
}
```

In the mapping we create, the key “::" holds the result of the call on our parent class (/obj/clothing) which does the same kind of thing for its parents, all the way along the line. Additionally, we add a motif entry to the mapping - this holds what our motif was set as. We don't need to call this method at any point, it gets called naturally as part of the saving and logon process.

Now we can log off and log on, and... it's still not there. Damnation! Why is this the case?

Well, we've saved the information fine. The second part of the auto load process is that each object is also responsible for restoring its own state. This works identically as a process, but we use the method init_dynamic_arg to do this. As with query_dynamic_auto_load, this method gets called automatically as part of the auto load process, and the parameters get provided from without. The first parameter is the mapping that contains all of the auto load information, and the second contains the player for whom the inventory is being loaded.

```
void init_dynamic_arg( mapping map, object ob) {
  if (map["::"]) {
    ::init_dynamic_arg(map["::"], ob);
  }
  if (map["motif"]) {
    set_motif(map["motif"]);
  }
}
```

In this first part of this code, we check to see if the mapping we have been provided has anything associated with the key “::". We know it does – that key contains all the information from the parent object, and so we call init_dynamic_arg on the parent passing only those parts of the mapping we got from it.

Motif is something we handle internally, so we check to see if one has been set, and if it has we call set_motif with whatever was in the mapping.

The fact that auto loading is a complex and tricky concept at the deep levels of the mudlib doesn't impact on your code especially. You just need to honour two rules in your objects:

- Make sure you honour the chain of invocation
- Store and restore your own state

If every object does this, then auto loading works like a charm and no-one has to know more than how to handle their own object state. It's really very clever when you think about what the alternatives would be.

Doing It Better

The above is how the majority of objects on the MUD handle auto loading. You'll see it a lot, so it's important you understand how it works. However, there's a better way that leads to much more readable, maintainable code – we use the add_auto_load_value and the query_auto_load_value functions to achieve our nefarious aims.

The first function, add_auto_load_value lets us add a piece of data to the mapping:

```
map = add_auto_load_value(map, "beautiful dress", "motif", query_motif());
```

The first parameter is the mapping we're working with,the second is a unique identifier for this object. The third is the name we'll set as the key of the mapping entry, and the last is the value itself. Thus, our method should look like this:

```
mapping query_dynamic_auto_load() {
  mapping map;

  map = ::query_dynamic_auto_load();
  map = add_auto_load_value(map, "beautiful dress", "motif", query_motif());
  return map;
}
```

And then we use init_dynamic_arg to put them back in place, like so:

```
void init_dynamic_arg( mapping map, object ob) {
  string tmp;
  ::init_dynamic_arg (map, ob);
  tmp = query_auto_load_value (map, "beautiful dress", "motif");
  set_motif (tmp);
}
```

You can see right away that the code is more readable, but there is an additional benefit – if every object in an inherit tree is using this system, what you get out the function is something that can be read and understood by a human. Alas, we don't use it all the way through the mudlib yet, and so results may vary from region to region. Nonetheless, this is how you should be handling your auto loading.

Back To Our Shop

Now we have a dress we can set a motif on, but we can't actually sell them with motifs because add_object doesn't let us set functions to be called on objects. However, there is a feature of item shops that allows us to provide fine-grained control over how objects should be created when someone buys them. We can provide a ‘create_object' method in our code – this will get called when a new object is to be created. It has the following form:

```
object create_object (string item) {    object ob;    return ob;  }
```

The string that the player chooses to buy goes into the function, and what comes out is the object we want the player to have. For example, let's say that no matter what, we wanted out player to get a delicious melon:

```
object create_object (string item) {
  object ob;
  ob = ARMOURY->request_item ("melon", 100);
  return ob;
}
```

Now when we list the stock, all this is for sale is melon:

```
The following items are for sale:
  A: a juicy melon for 2,66Gl (five left).
```

That's not good, what we want is to have a range of dresses with a range of motifs, like so:

```
add_object ("beautiful_dress with monkey motif", 3 + random (3));
add_object ("beautiful_dress with pirate motif", 3 + random (3));
```

And in our create_object, we handle the creating of the appropriate item, and set the appropriate motif in a switch statement:

```
object create_object (string item) {
  object ob;
  ob = clone_object (ITEMS + "beautiful_dress");

  switch (item) {
    case "beautiful_dress with monkey motif":
      ob->set_motif ("monkeys");
    break;
    case "beautiful_dress with pirate motif":
      ob->set_motif ("pirates");
    break;
  }
  return ob;
}
```

Alas, when we list them we get:

```
The following items are for sale:
  A: a beautiful blue dress for 1,0,0,0Gd (five left).
  B: a beautiful blue dress for 1,0,0,0Gd (three left).
```

There's no way for a player, short of browsing, to see the difference between the two. It's also a bit of a hassle to have the long string ‘beautiful dress with a blah motif' through our switch statement. If we're handling this directly with create_object, we have another format of add_object we might use, one that lets us set the display string as a third parameter:

```
add_object ("pirate dress", 3 + random (3),
  "beautiful dress with pirate motif");
add_object ("monkey dress", 3 + random (3),
  "beautiful dress with money motif");
```

It's the first value here that gets passed to our create_object, so that's what we need to base our switch on:

```
object create_object (string item) {
  object ob;

  ob = clone_object (ITEMS + "beautiful_dress");

  switch (item) {
    case "monkey dress":
      ob->set_motif ("monkeys");
    break;
    case "pirate dress":
      ob->set_motif ("pirates");
    break;
  }

  return ob;
}
```

You can also mix and match – create_object gets called when there is no object that can be cloned or requested from the armoury that matches the first string. If you add a valid object as the first parameter, you don't need to cater for it in create_object:

```
add_object (ITEMS + "beautiful_dress", 3 + random (3),
  "beautiful but unadorned dress");
```

Really, we'd need many more clothes here to justify the existence of the shop at all, but this is a tutorial and there's nothing to be gained by repeating the same content over and over again. We'll leave expanding the stock up to you.

Conclusion

Having items that can hold their state information is one of the most important things to be able to do when creating interesting code. Almost anything of any kind of complexity has a state that gets manipulated as the player works with it, and being able to store that state between logins is mandatory – there's no way to work around it other than resetting the state to some kind of starting value each time. That's bad for all sorts of reasons.

The auto load system is complex and intricate, and it requires all objects in the inherit chain to play nicely with all other objects. When that happens though, it's extremely flexible and puts the responsibility for proper restoring and storing of state information in the hands of individual creators, where it belongs.

Spelling It Out

Introduction

Remember how we set Lady Tempesre as a wizard and said we'd come back to her and give her some unique spells? Well, that's what we're going to do now. Coding spells and rituals is not strictly speaking something that most creators should be doing, but it's well worth knowing how they are put together and how they function. That's our topic for this chapter.

Most spells and rituals in the game exist in the /obj/spells and /obj/rituals directories, but this is a convention. There is nothing to stop us storing spells anywhere, and so we are going to keep ours in a sub-directory of betterville called ‘magic', like so:

```
#define   MAGIC           BETTERVILLE + "magic/"
```

So. with no further ado, let's a-do it!

The Anatomy of a Spell

At its simplest level, a spell is simply a set of configuration details along with code that happens when the spell succeeded, and code that happens if the spell fails. However, there is a set of minimal information we need to provide.

First of all, let's think of what we would like our Lady to do with a spell. We don't need to care if it's balanced or fair, because this is just a proof of concept. So, how about we be complete bastards and give her a spell to dump an enemy in the lair of the Beast? Oh yeah, that's the stuff.

First of all, we need to use the right defines in our code, and those come from magic.h. Let's begin our spell with the basic template, providing all of the necessary information about how the spell should actually behave:

```
#include <magic.h>

inherit MAGIC_SPELL;

void setup() {
  set_fixed_time( 1 );
  set_point_cost( 60 );
  set_power_level( 30 );
  set_directed( SPELL_DIRECT_LIVING );
  set_casting_time( 15 );
  set_name( "Lady Tempesre's Terrible Teleportation" );
  set_spell_type( "forces.offensive" );
  set_skill_used( "magic.spells.misc" );
  set_consumables( ({ }) );
  set_needed( ({ }) );
}
```

Quite a lot there – that's because spells are complicated and need to work in a structured way. Let's take each of the lines in turn and discuss what they do:

```
set_fixed_time( 1 );
```

As a caster's skills increase, the speed at which they can cast spells increases as well unless we set the time to be fixed. We can do this by giving this method a non-zero value. We do this with our NPC spell to ensure that our players have time to react before being dumped in the Beast's lair.

```
set_point_cost( 60 );
```

This is the amount of GP that the spell will cost when we cast it.

```
set_power_level( 30 );
```

This method indicates how powerful a spell is, which determines among other things the amount of mind-space it takes up and how much background magic it will deposit upon a casting.

```
set_directed( SPELL_DIRECT_LIVING );
```

This method allows us to set what kind of objects are valid targets for the spell. Direct doesn't mean here the same thing it does in add_command, it means that it can be used against living objects in the same room as the caster. The magic.h include file has all the other values to which this can be set.

```
set_casting_time( 15 );
```

This is how long the spell will take to cast – if we set the time to be fixed, this remains constant. Otherwise, it changes with the bonuses of the caster.

```
set_name( "Lady Tempesre's Terrible Teleportation" );
```

This is the ‘formal' name of the spell. Since this is a spell for an NPC, strictly speaking we don't need one. But why skimp? Why are you always skimping?

```
set_spell_type( "forces.offensive" );
```

This is the type of the spell. You should look at similar spells in /obj/spells/ to get a feel for what this should actually be.

```
set_skill_used( "magic.spells.misc" );
```

Later on in the spell, we'll choose the specific methods and levels needed for the casting. We use this to determine the bonus against which we'll check casting speed, as well as a general modifier for the power of the spell when it is successfully cast.

```
set_consumables( ({ }) );
```

With this function we can define spell components that get used up upon casting.

```
set_needed( ({ }) );
```

This function lets us define spell components that are needed but not consumed.

Once we've got all of that in place, we need to move on to the spell itself – each spell consists of a number of stages, each of which has messages that are shown to the caster and to the caster's environment. Each stage also sets the skills and bonuses needed:

```
set_ritual( ({
  ({
    ({
      "You form a claw with your hand.\n",
      "You form a claw with your hand.\n",
      "$tp_name$ forms a claw with $tp_poss$ hand.\n",
      "$tp_name$ forms a claw with $tp_poss$ hand.\n”
    }),
    "magic.methods.spiritual.conjuring",
    130,
    ({ })
  }),
  ({
    ({
      "You wiggle your fingers suggestively in the air.\n",
      "You wiggle your fingers suggestively in the air.\n",
      "$tp_name$ wiggles $tp_poss$ fingers suggestively in the air.\n",
      "$tp_name$ wiggles $tp_poss$ fingers suggestively in the air.\n"
    }),
    "magic.methods.physical.evoking",
    140,
    ({ })
  }),
  ({
    ({
      "You close one hand over the other.\n",
      "You close one hand over the other.\n",
      "$tp_name$ closes one hand over the other.\n",
      "$tp_name$ closes one hand over the other.\n",
    }),
```

```
    "magic.items.talisman",
    180,
    ({ })
  }),
  ({
    ({
      "You focus hard on your hands.\n",
      "You focus hard on your hands.\n",
      "$tp_name$ stares vacantly at $tp_poss$ hands.\n",
      "$tp_name$ stares vacantly at $tp_poss$ hands.\n"
    }),
    "magic.methods.mental.channeling",
    120,
    ({ })
  }),
  ({
    ({
      "You clap your hands together.\n",
      "You clap your hands together.\n",
      "$tp_name$ claps $tp_poss$ hands together.\n",
      "$tp_name$ claps $tp_poss$ hands together.\n"
    }),
    "magic.methods.mental.channeling",
    120,
    ({ })
  }),
}) );
```

Let's go over one stage of this spell in detail, since it illuminates what each stage of the spell does:

```
  ({
    "You form a claw with your hand.\n",
    "You form a claw with your hand.\n",
    "$tp_name$ forms a claw with $tp_poss$ hand.\n",
    "$tp_name$ forms a claw with $tp_poss$ hand.\n"
  }),
  "magic.methods.spiritual.conjuring",
  130,
  ({ })
```

First of all, the array of strings – these are the casting messages, broken down into the following:

```
({
  message to caster when caster is a target,
  message to caster when caster is not a target,
  message to room excluding all targets,
  message to targets excluding caster,
}),
```

We don't need to worry too much about this – unless you have a specific need based on the targeting of your spell, arguments 1 & 2 are usually the same, as are 3&4. You have the control there if you need it though.

Next in our stage comes the skill to use for this part of the spell, followed by the bonus to use for the taskmaster. Later on we'll see how we can have functions called at particular stages, or consume components at the right parts of the spell. We're aiming for a quick and simple piece of code first though.

Should the casting succeed, the method spell_succeeded gets called on our spell object, with the following parameters:

```
void spell_succeeded( object caster, object *targets, int bonus ) {  }
```

Here we can handle whatever we need to for the spell functioning properly. The first parameter is the caster, the second is an array of all the targets, and the third is the ‘bonus' with which the spell was cast – this is calculated from each of the skills used in the spell, and can be used as a general indicator of how powerful the spell casting was.

For our succeeded effect, let's give each target a chance to resist based on some skill. If they fail the taskmaster check, they get teleported to the beast. If they pass, nothing happens to them:

```
void spell_succeeded( object caster, object *targets, int bonus ) {
  int ret;

  foreach (object target in targets) {
    ret = TASKER->perform_task (target, "magic.methods.spiritual.abjuring",
      bonus, TM_FREE, 0);

    switch (ret) {
      case AWARD:
        tell_object (target, "%^BOLD%^%^YELLOW%^You feel more capable of "
          "resisting hostile magic!\n%^RESET%^");
      case SUCCEED:
      // Nothing happens.
        tell_object (target, "You feel the spell wash over you, but to no "
          "effect.\n");
        tell_room (environment (target), target->one_short() + " looks "
          "queasy for a moment, but nothing else seems to happen.",
          ({ target }));
      break;
      case FAIL:
        tell_object (target, "You feel your stomach lurch...");
        target->move_with_look (ROOMS + "beast_lair",
          "$N appear$s with a pop.",
          "$N disappear$s with a pop.");
    }
  }
}
```

When testing spells, we should always test them by casting them ourselves – NPCs don't give us fine-grained feedback, so we add the spell directly to our dusty cortex:

```
call add_spell ("tele", path_to_spell, "cast_spell") me
```

And then cast it on a poor, defenseless NPC for the fun of it:

```
> cast tele on lady
You prepare to cast Lady Tempesre's Terrible Teleportation on Lady Tempesre
  Stormshadow.
You form a claw with your hand.
You wiggle your fingers suggestively in the air.
You close one hand over the other.
You focus hard on your hands.
You clap your hands together.
Lady Tempesre Stormshadow disappears with a pop.
```

Cor, that's nice and effective. Too effective for a spell players get hold of, but fine for our proof of concept purposes.

We can also add a function to handle when a spell fails to work properly, to give our spell an Edge:

```
void spell_failed(object caster, object * targets, int bonus) {
  tell_object (caster, "Bugger....\n");
  tell_room (environment (caster), caster->one_short()
    + " looks panicked for a brief moment.\n", ({ caster }));
  caster->move_with_look (ROOMS + "beast_lair",
    "$N appear$s with a pop.",
    "$N disappear$s with a pop.");
}
```

Lower your skills sufficiently, and...

```
> cast tele on lady
You prepare to cast Lady Tempesre's Terrible Teleportation on Lady Tempesre
  Stormshadow.
You form a claw with your hand.
You wiggle your fingers suggestively in the air.
You close one hand over the other.
You focus hard on your hands.
You clap your hands together.

Bugger....

This is the lair of the Beast.  It is strewn with bones and other viscera.
There is one obvious exit: south.
The Beast is standing here.
```

So, as we can see – simple spells are simple!

More Complex Spells

However, we can also add a whole pile of complexity to spells to make them appropriately reactive. Spells often act as delivery engines for effects (which we will talk about later), but the engine itself has a number of ways it can be made more sophisticated. Let's add a second spell to our Lady, one that requires components for her to cast. She'll turn one of her dresses into an NPC that defends her for a minute or so before it disappears. However, she'll only be able to do it in her shop.

First, the basic setup:

```
void setup() {
  set_fixed_time( 0 );
  set_point_cost( 75 );
  set_power_level( 45 );
  set_directed( 0 );
  set_casting_time( 20 );
  set_name( "Lady Tempesre's Dancing Dresses" );
  set_spell_type( "forces.defensive" );
  set_skill_used( "magic.spells.defensive" );
  set_consumables( ({ "a beautiful blue dress" }) );
  set_needed( ({ }) );
  set_ritual( ({
    ({
      ({
        "You start to click your fingers.\n",
        "You start to click your fingers.\n",
        "$tp_name$ starts to click $tp_poss$ fingers.\n",
        "$tp_name$ starts to click $tp_poss$ fingers.\n",
      }),
      "magic.methods.physical.dancing",
      130,
      ({"#check_room"})
    }),
    ({
      ({
        "You do a little dance with a beautiful blue dress.\n",
        "You do a little dance with a beautiful blue dress.\n",
        "$tp_name$ does a little dance with a beautiful blue dress\n",
        "$tp_name$ does a little dance with a beautiful blue dress\n",
      }),
      "magic.methods.physical.dancing",
      200,
      ({ "a beautiful blue dress" })
    }),
    ({
      ({
        "You throw the dress up into the air.\n",
        "You throw the dress up into the air.\n",
        "$tp_name$ throws the dress up into the air.\n",
        "$tp_name$ throws the dress up into the air.\n",
      }),
      "magic.methods.spiritual.summoning",
      180,
     ({ })
   }),
  }));
}
```

Note here for stages one and two we're doing something slightly different – that which was an empty array for the earlier spell now contains some information. The #check_room element tells the code we want to call the function check_room to see whether or not the spell should progress beyond this stage. The method should return 1 on success and 0 on failure, like so:

```
int check_room(object caster, object *targets, class spell_argument args) {
  object env = environment (caster);

  if (!env) {
    return 0;
  }

  if (!env->query_lady_tempesre_clothes_shop()) {
    tell_object (caster, "This spell can only be cast in the boutique of "
      "Lady Tempesre.\n");
    return 0;
  }
  return 1;
}
```

The second example is on stage two, and it's what tells the spell to consume our component at that point. If we don't have the component any more, it'll quit the spellcasting there.

Once our spell has finished casting then we have our succeeded and failed functions as before:

```
void spell_succeeded( object caster, object *targets, int bonus ) {
  object ob = clone_object (CHARS + "dancing_dress");
  ob->move (environment (caster), "$N flutter$s around.",
    "$N disappear$s with a pop.");
  caster->add_protector (ob);
}

void spell_failed( object caster, object *targets, int bonus) {
  tell_room ("Nothing happens in a most spectacular fashion.\n");
}
```

The NPC referenced here is a simple ‘dress' NPC created just to be summoned by the spell:

```
#include "path.h"
inherit INHERITS + "betterville_npc";

void setup() {
  set_name ("dress");
  set_short ("beautiful blue dress");
  add_adjective (({"beautiful", "blue"}));
  basic_setup ("elemental", "fighter", 50 + random (10));
  set_long ("This is a beautiful blue dress.  It appears to be... "
    "mobile without visible support.\n");
  call_out ("do_death", 30);
}
```

```
object make_corpse() {
  tell_room (environment (this_object()), one_short()
    + " explodes in a spray of silk!\n");
  return 0;
}
```

There is a slight difference here in that we provide a make_corpse method – this overrides the basic functionality in the death code and ensures that we don't end up with corpses of blue dresses hanging around. Instead, we just get a message when it dies, and not a corpse:

```
The beautiful blue dress explodes in a spray of silk!
```

We return 0 to indicate that no corpse is forthcoming. You can also return an actual object (for example, one of our original blue dresses!) if you want something else to take the place of a dead creature. Note too that each comes with an expiry date – they get a call_out to do_death thirty seconds after they are created. They only last for a short period of time.

Now all we have to do is bind these spells into Lady Tempesre, and she'll have them to hand for when she is attacked.

Spellbinding

First, we give her the spells in setup, in the same way we gave them to ourselves:

```
add_spell ("teleport", MAGIC + "tempesre_teleport", "cast_spell");
add_spell ("dresses", MAGIC + "dancing_dress", "cast_spell");
```

Then, we create combat actions for each, in the style that we did with the beast:

```
add_combat_action (25, "cast dresses", ({"cast_dresses"}));
add_combat_action (10, "cast teleport", ({"cast_teleport"}));
```

And then we add the functions to handle the core of the spellcasting:

```
void cast_dresses(object attacker, object defender) {
  if (!spare_dress) {
    tell_room (environment (this_object()), one_short() + " grabs a dress "
      "from one of the racks.\n");
    spare_dress = clone_object (ITEMS + "beautiful_dress");
    spare_dress->move (this_object());
```

```
  }

  init_command ("cast dresses", 1);
}

void cast_teleport(object attacker, object defender) {
  object enemy;
  if (attacker == this_object()) {
    enemy = defender;
  }
  else {
    enemy = attacker;
  }
  init_command ("cast teleport on " + file_name (enemy), 1);
}
```

Unfortunately, this will not work quite as desired – NPCs have the capability of casting multiple spells at the same time, and we can't guarantee our spell-casting will give suitable time intervals between casts. We can resolve this in several ways, but the way that works easiest for the second problem is to put a kind of ‘cooldown' on castings, like so:

```
void cast_dresses(object attacker, object defender) {
  if (query_property ("cast dresses")) {
    return;
  }
  ...
  init_command ("cast dresses", 1);
  add_property ("cast dresses", 1, 15);
}

void cast_teleport(object attacker, object defender) {
  object enemy;
  if (query_property ("cast teleport")) {
    return;
  }
  ...
  init_command ("cast teleport on " + file_name (enemy), 1);
  add_property ("cast teleport", 1, 30);
}
```

In this way, we can easily manage the speed of spell-casting and at least give people a chance to defend against her wrath. Together, these two spells make her capable of dealing with those who see fit to disturb her commerce. However, this doesn't stop her from casting two spells at once. That we can deal with by adding in a call to query_casting_spell in each function, like so:

```
if (query_casting_spell()) {
  return;
}
```

We put this in both our combat actions, and our Lady is now prepared to defend herself fairly with her nifty bespoke magics, like so:

```
The upwardly mobile socialite moves aggressively towards Lady Tempesre
Stormshadow!

The upwardly mobile socialite punches at Lady Tempesre Stormshadow but she
easily dodges out of the way.

The upwardly mobile socialite punches at Lady Tempesre Stormshadow but she
easily dodges out of the way.

Lady Tempesre Stormshadow forms a claw with her hand.

The upwardly mobile socialite exclaims: Stop, this is part of a different
story!

Lady Tempesre Stormshadow wiggles her fingers suggestively in the air.

Lady Tempesre Stormshadow closes one hand over the other.

The upwardly mobile socialite jabs Lady Tempesre Stormshadow in the right
arm.

The upwardly mobile socialite tickles Lady Tempesre Stormshadow in the right
hand.

Lady Tempesre Stormshadow stares vacantly at her hands.

The upwardly mobile socialite jabs Lady Tempesre Stormshadow in the right
arm.

Lady Tempesre Stormshadow claps her hands together.

The upwardly mobile socialite jabs Lady Tempesre Stormshadow in the chest.

The upwardly mobile socialite kicks Lady Tempesre Stormshadow in the neck.

The upwardly mobile socialite disappears with a pop.
```

Let that be a lesson to other dozy girls to follow – don't mess with Our Lady.

Conclusion

The construction of a spell is a task that not all creators should be considering – they can't go into the game without the say-so of those in Guilds, and usually there are more ideas for spells than there are actual needs for one. However, we should still know how they are put together so that we can understand the nature of the process.

Having NPCs that intelligently use spells however is a thing all creators should be able to do, and the interaction of components, casting times and spell stages requires an extra level of care and consideration when building the combat actions of our NPCs - especially when our NPCs are using bespoke spells for colour and flavour in our areas.

Cause and Effect

Introduction

In the last chapter we looked at spells, but rather simple spells. Often spells act as a kind of delivery engine for an effect rather than something that instantly delivers a payload. In this chapter we are going to discuss the idea of an effect, how it can be used, and what it can do.

Once again, we start by adding a new define to our path.h file:

```
define   EFFECTS         BETTERVILLE + "effects/"
```

We are going to address the path that leads to our library in this chapter – as you'll undoubtedly remember from Being A Better Creator, this is supposed to be hidden. That's no problem, but we're going to make the searching a little more interesting than we have done in the past.

What's An Effect?

The easiest way to think of an effect is as a temporary set of conditions that impact on a player or an object. For example, an effect may be a disease, or a curse, or a buff, or any number of other things. They get attached to objects on a conditional basis – they may last until a certain condition is met, or until a certain period of time has passed. While they are attached to an object, that object can be impacted on an ongoing basis.

In the Olden Days, effects often used to come as a pair with a thing called a shadow. A shadow is an object that can be attached to another object and works like a temporary mask on a function. We might have a shadow that overrode adjust_xp to give double the experience it would normally give, or overrode query_skill_level to simulate an object that gave bonuses to certain skills.

We don't do that any more – that particular mudlib feature is now considered to be deprecated. You'll still see a number of shadows around and about because we haven't gotten them all out of our code. **No new shadows should be coded.**

However, that doesn't mean effects are useless – far from it.

What we're going to do in this chapter is build an effect that acts as ‘damage over time' system on a player. As a result of searching for the hidden path, they'll get scratched and damaged and bleed for a while. It's a simple effect, but that that will illustrate a good deal about how they are put together.

Effects don't need to inherit anything – they exist as self-contained sets of code. However, they must implement five functions to work properly.

```
#include <effect.h>

int beginning( object player, mixed args, int id ) {
}

int merge_effect( object player, mixed old_args, mixed new_args, int id ){
}

void end( object player, mixed args, int id ) {
}

string query_classification() {
}

void restart( object player, mixed args ) {
}
```

The first of these is *beginning*, and it gets called when an effect is first added to another object. The *merge_effect* function is called if an effect is added to an object that already has the same effect active. The *end* function gets called when the effect is removed from an object. *Restart* is called when a player logs off and then on again. Finally, *query_classification* provides an 'identifier' code.

Let's start by filling in a placeholder for each of these functions and discussing what their parameters do.

```
int beginning( object player, mixed args, int id ) {
  tell_object (player, "Argh!  You must have put your hands in some "
    "stinging nettles.\n");
  player->adjust_hp (-1 * args);
  return args;
}
```

The first parameter is the object on which the effect has been applied – it doesn't have to be a player, but since this particular effect will indeed be applied to players, we may as well indicate that with the parameter name. When we add an effect to an object, we can provide some arguments to it for configuration purposes. Within the beginning function, they are accessible as the second parameter. The third argument you don't need to worry about.

When we return a value from this function, the value we return becomes the ‘argument' of that effect – we can access that later in code, which we will come back to later. For now, all this effect does when it is applied to a player is do an amount of damage equal to the argument we pass it.

The end function doesn't have to do anything, but it's always called when an effect is removed and it can be a good way for us to provide some information to the player reflecting that the effect has come to an end. The parameters are the same as for beginning:

```
void end( object player, mixed args, int id ) {
  tell_object (player, "The stinging from the nettles subsides.\n");
}
```

For query_classification, we give some meaningful identifier for this effect. It doesn't really matter what it is, but for convention we usually use something like ‘body.nettles' to give a fair idea of what it is and where it comes from:

```
string query_classification() {
  return "body.nettles";
}
```

By default, if two effects are added to the same object, the object gets two copies of an effect running on them. Usually we don't want that, and so we override that behaviour with the use of merge_effect. Whatever we return from this function becomes the new value of the argument of the effect. Let's make it so that we simply use the stronger of the two:

```
mixed merge_effect( object player, mixed old_args, mixed new_args, int id ) {
  if (old_args > new_args) {
    return old_args;
  }
  return new_args;
}
```

Restart gets called each time the effect gets restarted, so let's use it to supply a message to the player:

```
void restart( object player, mixed args ) {
  tell_object (player, "Argh!  You must have put your hands in some "
    "stinging nettles.\n");
}
```

And that, believe it or not, is an effect. It's not much of one, but it's an effect nonetheless. We can test it out by using the add_effect method, like so:

```
call add_effect ("/w/drakkos/betterville/chapter_14/effects/nettles", 100) me

Argh!  You must have put your hands in some stinging nettles.
```

Along with that message comes 100 points of damage, as we planned all along. To get a list of all the effects on an object, I can use the ‘effect' command:

```
> effect drakkos

Effects on Drakkos:
[0] body.wetness (56)
[1] body.nettles (100)
```

In square brackets before each of these effects is a numeric identifier, the effect number (or **enum** for short). Here we can see the effect with the enum 1 is body.nettles, which is what we gave it for a classification. The 100 in brackets indicates what the argument of that effect is. This effect will remain in place until I die, or I delete it like so:

```
> call delete_effect (1) me
The stinging from the nettles subsides.
```

I can add this effect as many times as I like to myself, and I will only ever have one body.nettles effect on me – merge_effect deals with that.

So that's an effect – it's simple, but not exciting. Let's open this puppy up though and see what we can do.

The Power of Effects

The real power of effects comes from how easily we can cause our code to have certain functions called at regular intervals. With our nettle effect, it just does the damage once and that's it. If we want damage to also be done on a periodic basis, we can do that using the submit_ee (ee is short for 'effect event') function on our player. This function takes three parameters – the first is the function to be called when the effect event occurs. The second is the interval between the ticks of the event, and the third is how often the event should be submitted. This should be one of three values, as defined in effect.h:

Effect Interval	Description
EE_REMOVE	Call it once, and then remove the effect from the player when it has occurred.
EE_ONCE	Call it once, but leave the effect intact
EE_CONTINUOUS	Call it repeatedly, leaving the tick period between invocations

So if we wanted damage done continuously while the effect was active, that's very easily done – the function we tell the effect to trigger comes with the same parameters as for beginning and end:

```
mixed beginning( object player, mixed args, int id ) {
  tell_object (player, "Argh!  You must have put your hands in some "
    "stinging nettles.\n");
  player->adjust_hp (-1 * args);
  player->submit_ee( "damage_me", 20, EE_CONTINUOUS );
  return args;
}

void damage_me(object player, mixed args, int id ) {
  player->adjust_hp (-1 * random (args));
  tell_object (player, "Oooh, that stings!\n");
}
```

Every twenty seconds now, we'll trigger the damage_me function and that will do a random amount of damage to the player based on how large an initial argument we set. It will do this until the player dies, which is perhaps not exactly fair. To make it a little less cruel, we combine this with a submit_ee that has the interval set to be EE_REMOVE. This doesn't even need to have a function, it'll just cause the effect to be automatically removed when the tick period has passed.

```
mixed beginning( object player, mixed args, int id ) {
  tell_object (player, "Argh!  You must have put your hands in some "
    "stinging nettles.\n");
  player->adjust_hp (-1 * args);
  player->submit_ee( "damage_me", 20, EE_CONTINUOUS );
  player->submit_ee( 0, args, EE_REMOVE);
  return args;
}
```

Now we have an effect that is time limited by the initial argument too. If we wanted, we could make the argument more granular so that we added an array instead of a single integer, but the theory is the same.

Now the last thing we need to do to this effect is make it merge properly. It's already handling the possibility of a player being affected by a stronger version of the effect (remember, whatever we return for merge_effect becomes the new argument of the effect), but we should also make sure it impacts properly on the duration. That's easy to do with the expected_tt function, like so:

```
mixed merge_effect( object player, mixed old_args, mixed new_args, int id ) {
  int time_left = player->expected_tt();
  time_left += new_args;
  player->submit_ee (0, time_left, EE_REMOVE);
  if (old_args > new_args) {
    return old_args;
  }
  return new_args;
}
```

Only one EE_REMOVE is ever active on a player for a particular effect, so when we submit our new one we overwrite the old one. Now, each new application of the nettles effect will increase its duration, but not cumulatively impact on its strength. For a real effect, we probably don't want this either (some set ceiling is usually appropriate, although we don't always honour that), but it'll do for us.

The next step is to write some code that adds this simple effect to a player. We go to Betterville_02 to do this. First we modify the exit to the trail so it isn't obvious, and then we create a do_search function to catch player use of the search command:

```
#include "path.h"
#include <tasks.h>

inherit INHERITS + "outside_room";

void setup() {
  set_short ("skeleton room");
  add_property ("determinate", "a ");
  set_long ("This is a skeleton room.  There is some brush here.\n");

  add_item ("brush", "It looks unpleasant, full of nettles and "
    "other horrible stinging things.");
  add_item (({"nettle", "horrible stinging thing"}), "You wouldn't "
    "want to take a widdle in that lot.");

  set_light (100);

  add_exit ("north", ROOMS + "betterville_03", "road");
  add_exit ("southwest", ROOMS + "betterville_01", "road");
  add_exit ("southeast", ROOMS + "betterville_08", "road");

  modify_exit ("southeast", ({"obvious", 0}));
}

int do_search (string str) {
  int found;
  int success;

  success = TASKER->perform_task (this_player(), "other.perception",
    100, TM_FREE);

  switch (success) {
    case AWARD:
```

```
      tell_object (this_player(), "%^YELLOW%^%^BOLD%^You feel a little more "
        "perceptive.\n%^RESET%^");
    case SUCCEED:
      tell_object (this_player(), "You see a trail leading through "
        "the brush to the southeast.\n");
      found = 1;
    break;
    case FAIL:
     found = 0;
    break;
    default:
  }




  if (random (2)) {
    this_player()->add_effect (EFFECTS + "nettles", 100 +
      random (100));
  }

  if (found) {
    return 1;
  }
  else {
    return -1;
  }
}
```

It is the add_effect method that provides us with a mechanism for applying the effect, and that method is common to all objects that inherit /std/object along the way. As such, effects can exist on inanimate objects as easily as they can on players, and the mechanisms for dealing with them are identical.

Working With Effects

Coding effects is one part of the problem. The next is how we can work with effects that are already on players. For example, we may wish to have different courses of action available to people who are afflicted with particular effects.

The first thing we commonly want to do is determine if a player has a particular effect on them. We do this with the effects_matching function to which we pass the classification of effect for which we want to look. As a result, we get an array of all the matching effect numbers (enums, remember), or an empty array if no effects match the classification.

Let's go back to our search function – let's tell players afflicted with our effect that it hurts too much for them to search in the room:

```
int do_search (string str) {
  int found;
  int success;
  int *effects;

  effects = this_player()->effects_matching ("body.nettles");

  if (sizeof (effects)) {
    tell_object (this_player(), "You are still stinging from your "
      "brush with the nettles, and you can't bring yourself to search "
      "too effectively.\n");
    return 0;
  }

  ...
}
```

The next thing we often want to do is remove effects based on some external factor. An old folk remedy to nettles claims that the plant 'horsetail' is effective for removing the stinging sensation. Let's add some horsetail here and have its impact be to remove the effect if it exists. We use delete_effect to handle that. First, the add_item:

```
add_item ("horsetail", (: horse_tail_function :));
```

And then the function that does the magic:

```
string horse_tail_function() {
  int *effects;
  string ret;

  ret = "Yes, there is some horsetail here.  ";

  effects = this_player()->effects_matching ("body.nettles");

  if (sizeof (effects)) {
    ret += "Throbbing from the stinging nettles, you clutch it "
      "thankfully.";

    for (int i = 0; i < sizeof (effects); i++) {
      this_player()->delete_effect (effects[i]);
    }
  }

  return ret;
}
```

Note that we delete the effects in a loop – remember an effect without a merge_effect will result in more than one effect being applied to the object, and we can't guarantee that the effects we are working with have only one instance set on them. Whenever deleting an effect then, we do it properly – we get rid of all of them.

Now, the thing about folk cures is that they are horribly ineffective. Really, our horsetail should only reduce the impact of the effect, rather than remove it entirely. To do that, we need to know what the value of the effect actually is. We can get the current argument with arg_of, and we can change its value with set_arg_of. Both of these need us to provide the enum of the effect we wish to query or modify, like so:

```
string horse_tail_function() {
  int *effects;
  string ret;
  int val;

  ret = "Yes, there is some horsetail here.  ";

  effects = this_player()->effects_matching ("body.nettles");

  if (sizeof (effects)) {
    ret += "Throbbing from the stinging nettles, you clutch it "
      "thankfully.";

    for (int i = 0; i < sizeof (effects); i++) {
      val = this_player()->arg_of (effects[i]);
      val = val - 25;

      if (val <= 0) {
        this_player()->delete_effect (effects[i]);
      }
      else {
       this_player()->set_arg_of(effects[i], val);
      }
    }
  }
}
```

For complicated effects, the arguments may well be more complex than just an integer, but the principle is identical. The arg_of function gives us what the argument of the effect is, and set_arg_of allows us to change it.

Bits and Pieces

There are some other details that we need to talk about before you're ready to make use of effects properly. The first is – when you're writing an effect, you cannot use this_player. It is hardly ever going to be what you want it to be. Instead, use the object reference provided as a first parameter to all the methods – this ensures that you're working with the object to which the effect is attached.

Sometimes we want effects that exist indefinitely – it's entirely possible that we may want an effect that doesn't have any persistent behaviour associated with it (as in, no events get scheduled). The effects management code can detect effects that have no pending events, and will often remove them. If we are creating a genuinely indefinite effect, we need to stop it doing this by adding a query_indefinite function:

```
int query_indefinite() {
  return 1;
}
```

Upon death, all effects are removed from a player. If we have an effect that should persist through death, we also include a survive_death function:

```
int survive_death() {
  return 1;
}
```

If we are creating an effect that a player may be able to cure (with a spell, ritual, or such), we can define a test_remove function to control whether our effect is actually impacted. It takes four arguments – the object on which the effect is active, the argument of the effect, the ID of the effect (ignore this), and the bonus with which you should work with to see if an effect should be removed. If we return a non-zero value, the effect gets deleted when the appropriate trigger condition occurs. If we return 0, the effect remains in place:

```
int test_remove(object player, mixed arg, int id, int skill_bonus) {
  if (skill_bonus > arg) {
    return 1;
  }
  return 0;
}
```

Finally, remember that many of the effects that you will read examples of also use shadows. We don't do that kind of thing any more – shadows were very powerful, but came at considerable cost to the MUD's efficiency. If there is functionality you want to create that would require a shadow object, don't go ahead and write one – instead, consult a more experienced creator who might be able to put forward an alternate course of action.

Conclusion

Effects are a tremendously useful way of creating temporary conditions that get attached to objects in the game. With then we can create all kinds of sophisticated behaviours such as poisons, buffs, debuffs, and all sorts of things in between. While a lot of their ‘oomph' has been taken away with the deprecation of shadows, you'll still find them a useful tool to have in your toolbox. Many of our spells and our rituals act as a delivery mechanism for effects, and much of our more intricate code makes heavy use of them. .

Here we have created a relatively simple effect, but with a little modification it would be possible to change it into a powerful ‘heal over time' spell, or a more comprehensive ‘damage over time' that incorporated all sorts of deleterious effects. Have a read over what we have available in /std/effects/ and see the kind of things that you can do with a little bit of imagination.

Function Pointers

Introduction

The last Betterville-related thing we're going to cover in this book is the topic of function pointers. We've been using these for a while now, and the final step that we are going to take is to explain what they are, how they work, and what they can do for your code. There's nothing left for us to add to Betterville at this point (except for some things I shall suggest as reader exercises), all we're doing is explaining a few of the things we've had to take for granted up until this point.

The Structure of a Function Pointer

All the way through our code we've been using variables, and these are just containers for some information that we want to store. A function pointer is a variable too, a variable of type function. However, unlike the other data types which are passive, a function pointer contains some actual code that can be executed (or evaluated) on command. Function pointers of the type we've been working with are what I like to refer to as ‘happy code' because they're enclosed in smiles like so:

```
function f = (: 1 + 2 :);
```

One way to think of a function pointer is as a ‘delayed blast function'. It sits there until someone decides it's time for the code to be triggered through the use of the **evaluate** efun:

```
return evaluate (f);
```

Now, this example is pretty trivial – the real power of function pointers comes from how flexible they are. We can make use of placeholder variables inside a function pointer like so:

```
function f = (: $1 + $2 :);
```

Then when we evaluate the function, we can pass parameters to it. They get handled in the pointer in the order in which they are provided:

```
return evaluate (f, 10, 20);
```

When the function gets evaluated, 10 gets substituted for $1, and 20 for $2.

We can bind the function pointer to a local function – this is something we've done quite a lot of with add_items. Look back at our horsetail add_item in betterville_02:

```
add_item ("horsetail", (: horse_tail_function :));
```

What we've done here is create a function pointer that binds to the locally defined function called horse_tail_function. When this function pointer is evaluated, it calls the defined function. We can even do this with arguments, if we so desire:

```
int test_function(int a, int b) {
  return a + b;
}

int test_pointer() {
  function f = (: test_function, 10, 20 :);
  return evaluate (f);
}
```

All of this works perfectly. We can also define function pointers that look, to all intents and purposes, like normal function calls:

```
function f = (: test_function ($1, $2) :);
```

These kind of function pointers cause problems with debugging though – when an error occurs in one of these, the runtime trace tells us simply that there is a problem with the pointer, but very little useful information beyond that.

The last kind of function pointer lets us simply write a function and store it in a variable. This is horrible in all sorts of ways, so please don't do it. However, you may find other people doing it in other parts of the mudlib, so you should at least be able to recognize it when it occurs:

```
int test_pointer() {
  function f = function (int a, int b) {
    return a + b;
  };
  return evaluate (f, 10, 20);
}
```

If I see you doing this anywhere, I will cut you. You have been warned!

All of these are different ways of achieving the same end – create a ‘delayed blast function' that gets evaluated at a later date. The power of this as a mechanism can't really be over-stated – it lets you bundle an awful lot of functionality into a very small space if you know what you're doing. To see the power invested in this kind of data type, we're going to talk about what I like to refer to as the Four Holy Efuns.

The Four Holy Efuns

There are four efuns that, when used in combination with function pointers, will make available a huge amount of functionality with very little expended effort. They are implode(), explode(), filter(), and map(). The first two we have seen. The last two we will talk about in depth in this chapter.

Filter is an efun that takes an array, and then returns another array that contains only those elements that passed a particular check. This check can be defined as a local function, or implemented as a function pointer. Within the function pointer, the argument $1 refers to the ‘current element' in the same way our variable in a foreach loop refers to the current element of the array we are looping though. Let's say for example that we wanted to get the list of online creators. One line of code can do that:

```
return filter (users(), (: $1->query_creator() :))
```

The array we provide to the efun is the object array returned from the users() function. We step over each element of that array, calling query_creator on each. Those objects that return 1 from that are added to the array to be returned. Those that return 0 are discarded.

Do you want to get the list of interactive objects in the room in which you are currently standing? Try this in an exec:

```
exec return filter (all_inventory (environment (this_player())),
  (: interactive ($1) :))
```

The process is exactly the same – filter steps over each element of the array and passes it as an argument in the interactive efun. Those objects that return 1 get returned from the filter.

The sister function of filter is map, and it works in the same way – the difference is, it doesn't filter out the objects, it gives the return value of whatever function we provided it. Would you like to get the names of all players online? Well, you can do this:

```
return map (users(), (: $1->query_name() :))
```

The process that map goes through is to get the array provided by users, call query_name() on each element, and take the result of that function call and add it to an array to be returned. What we get out of that then is an array containing all the names of each online player.

Now, here's where we start getting a bit adventurous. The real power of these efuns comes from when we combine them together. What if you want the names of all online creators? Really that's the process we went through above, except the result of one feeds into the other:

```
string *online_creator_names() {
  object *creators = filter (users(), (: $1->query_creator() :));
  return map (creators, (: $1->query_name() :));
}
```

You will often find these chained together, especially if you need to quickly check something with an exec:

```
exec return map (filter (users(), (: $1->query_creator() :)),
  (: $1->query_name() :))
```

Rapidly, code like this becomes very difficult for people to effectively read. Function pointers give a huge amount of expressive power with a minimum of coding effort, but the cost is in casual readability of your code. Nonetheless, the benefits are hard to deny.

Implode can also take a function pointer as its second parameter. When we do this, it takes the first and the second parameters of the array we are imploding, and passes them into the function. It then takes the result of that and the next element, and passes them in, and so on. So imagine if we had an array of integers, like so:

Index	Element
0	10
1	20
2	30
3	40

And now imagine that we ran the following implode over it:

```
implode (arr, (: $1 + $2 :));
```

For the first step of the implode, the function gets elements 0 and 1:

```
10 + 20
```

The result of this is 30, and at the next step of the implode, the function gets the results of the previous evaluation (30), as well as the next element to be imploded (element 2):

```
30 + 30
```

And then at the last, it gets the results of the evaluation plus the last remaining element:

```
60 + 40
```

The result of this implode is to return the value 100 – it sums up all of the elements in the function pointer in a quick, elegant. way.

Explode doesn't permit the use of a function pointer, but it's the natural companion to implode as thus still remains one of the Four Holy Efuns.

Taking the four of these together gives tremendous expressive power to a creator when dealing with strings or arrays. For example, let's say I have the following string:

```
"drakkos,taffyd,wodan,sojan,dasquian"
```

And what I want is to check to see if these people are online, and if they are display their properly capitalized names. I can do that with a function:

```
string fix_names (string str) {
  string *names = explode (str, ",");
  string *online = ({ });
  string ret;

  for (int i = 0; i < sizeof (names);i++) {
    if (find_player (names[i])) {
      online += ({ cap_words (names[i]) });
    }
  }
  ret = implode (online, ", ");
  return ret;
}
```

Or I can do it in a much more compact way with the four holy efuns:

```
return implode (map (filter (explode (str, ","), (: find_player ($1) :)),
  (: cap_words ($1) :)), ", ");
```

You will find much of our code is based around this kind of compact representation. However, this should all come with a disclaimer – there is very little that you can do with function pointers that you cannot do with explicit functions, and these are always more readable and maintainable. Working effectively with function pointers is the key to understanding lots of the mudlib, and to making your exec command the best friend you ever had, but there are only a few situations in which you should really use them. Many of us over the years have gotten into the bad habit of using them largely automatically, and you would be doing yourself a favour if you used them only sparingly.

However, we are now in a position to look at the areas where we have used function pointers in our code, and why we have gone down that route. Throughout Betterville (and indeed, Learnville), function pointers were used only when they were the only way to achieve the necessary ends.

Back To The Library

Look at our library – it's beautiful. It is absolutely bursting with quests, and we should be proud of ourselves that we have them. However, one last thing remains – the long description of the library:

```
set_long ("This is the main part of the library.  It's a mess, with "
  "discarded books thrown everywhere!\n");
```

Here, we need to have something a little more dynamic – it doesn't make sense to have this long description when the books have all been sorted away. Instead, the long description should change with the state of the library. Now, if we have a long description that can change in the course of play, we can make a dynamic long description by making use of a function pointer. There are other ways to do it, but they are awkward. All we do is something like this:

```
set_long ((: library_long :));
```

And then a function to return a string based on the state of the library:

```
string library_long() {
  string ret = "This is the main part of the library.  ";

  if (check_complete()) {
    ret += "It is immaculate, with books neatly stored on shelves. ";
  }
  else if (!sizeof (query_unsorted_books())) {
    ret += "It is very tidy, but it doesn't look as if the books have "
      "been sorted properly.";
  }
  else {
    ret += "It's a mess, with discarded books thrown everywhere!";
  }

  return ret + "\n";
}
```

Now, this is the only good way of providing a dynamic long – but there are other options. If the state of the library is going to change a lot, then it's fine. If it's going to change once, then consider simply setting the long to be something different at a later part of the code.

Internally, the set_long that we give the room gets stored as the variable long_d in /std/basic/desc. When the long description is queried (for example, when we do a look in a room), the following function gets triggered:

```
varargs mixed query_long(string str, int dark) {
  if (functionp(long_d))
    return evaluate(long_d);

  return long_d;
}
```

This is the difference between setting a function pointer as the long, and setting the return value of a function. For example, consider the following:

```
set_long ( library_long() );
```

This is evaluated only when the set_long is called – whatever comes out of that function is what becomes the string stored in long_d. As such, this just creates a static description – the only difference is where that static description comes from. If we store it as a function pointer it means that every time the query_long function is called, the function pointer gets evaluated. As I'm sure you can imagine, that's not great for the efficiency of your code if it's not necessary, especially if your function requires a lot of CPU time to process. We can take that hit every now and again if there is suitable benefit, but it's not something to get into the habit of doing.

It is exactly this same principle at work when we set a function pointer to be evaluated in an add_item:

```
add_item ("horsetail", (: horse_tail_function :));
```

That function gets evaluated each time the player looks at the horsetail add_item. Again, it comes at a cost, but we can take that hit provided it's not over-used.

Function Pointer Support

Function pointers are not supported universally throughout the mudlib – they require someone to have put the support in place in the low level functions. However, they are supported widely. For example, if you wish to change the name of a function attached to an add_command, you can do that with a function pointer. Let's say we wanted to give players the option to either ‘read' our books or ‘browse' them, with the same thing happening in either case. We simply use a function pointer as a third parameter to add_command:

```
add_command ("read", "blurb on <string'book'>");
add_command ("browse", "blurb on <string'book'>", (: do_read :));
```

If we use a function pointer in this way, we always need to prototype the function because the driver will do compile time checking on the code to make sure the function is present.

We can even use function pointers to simplify the parameter list of the methods themselves. For example, we don't care about anything other than the blurb the player entered for the do_read function, and yet we still have a pile of parameters we need to represent:

```
int do_read (object *indirect_obs, string dir_match, string indir_match,
  mixed *args, string pattern) {
}
```

We can use the function pointer to say ‘actually, all I want is the first element of the fourth parameter':

```
add_command ("read", "blurb on <string'book'>", (: do_read ($4[0]) :));
add_command ("browse", "blurb on <string'book'>", (: do_read ($4[0]) :));
```

And now, rather than our complicated list of parameters, we just have one string coming in:

```
int do_read (string book) {
}
```

We do that a lot around these here parts. As long as you're clear in your mind as to what the little dollar sign numbers mean, it should be pretty simple for you to work out what information is going into the functions.

Often, we use a different kind of way of supporting functions that are dynamically called from other functions. Where possible, you should use these rather than pointers. A case in point is in a load_chat or add_respond_to_with, such as with our Beast:

```
load_a_chat(120,  ({
  1, "#animal_growling",
  1, "@roar",
}));

add_respond_to_with( ({
    "@say",
    ({"lilith"}),
  }),
  "#lilith_response");
```

These are handled in a different way in the mudlib. They are not function pointers, they get parsed by lower-level mudlib inherits to do the right thing. Really all that happens is when this chat or response is selected, the function checks to see if the first character of that response is a #, and if it is it uses the function call_other to trigger the function response. This is taken directly from expand_mon_string in /obj/monster:

```
switch ( str[ 0 ] ) {
  case '#' :
  args = explode(str[ 1..], ":");
  call_other( TO, args[0], args[1..]... );
break;
```

The # system is supported erratically throughout our mudlib, but you should use it instead of a function pointer whenever you can. Function pointers are tricky for others to read unless they are thoroughly steeped in sin, difficult to debug in certain cases, and do not benefit from any specialized functionality that comes from more tailored support in the mudlib.

A Few More Things About Function Pointers

You can pass parameters to a function pointer, but when doing so you can't make use of local variables:

```
string *online_creator_names(string is_online) {
  object *creators = filter (users(), (: $1->query_name() == is_online :));
  return map (creators, (: $1->query_name() :));
}
```

This will not work – you'll get a compile time error saying ‘You can't use a local variable in a function pointer'. If you want to use a variable like this, you need to explicitly force them to be literals rather than variables by enclosing them in $(), like so:

```
object *creators = filter (users(), (: $1->query_name() == $(is_online) :));
```

This syntax tells the MUD 'hey, I'm just going to need the value of this - extract it from the variable, because the variable might not be around later'.

You will also find that sometimes function pointers cause all sorts of strange errors. Function pointers are bound to a particular context (a specific object), and they don't save with save_object or work very well in call_outs. The error you'll see in this case is along the lines of ‘Owner of function pointer is destructed', and it will cause a runtime.

Conclusion

Powerful as all get out, but costly in terms of code readability and often efficiency – function pointers are the guilty little indulgences of Discworld. We have comprehensive (if not universal) support for them in our mudlib, and they are the key to making your exec command a powerful weapon rather than a distracting novelty. It is important that you know how to use them, because there are very few parts of our game that aren't touched by them in some way, and many of the things you might like to do require at least passing familiarity with the concept. Think way back to LPC For Dummies 1 – we couldn't even keep them out of the introductory text for Discworld.

Many of us use them automatically, but that's not a good thing. They should be used with caution and with forethought. Whenever it is possible, use good, honest, properly written functions rather than function pointers. It may cost you a little extra time, but those who have to maintain your code will thank you for it.

Achieving Your Potential

Introduction

Achievements are a new mechanism for Discworld MUD - a way to get an extra degree of 'oomph' out of your code without you needing to actually write anything new. Achievements give you a way of focusing attention on the unique features of your development, and rewarding those players who participate most fully in the gaming experience you have provided.

Unlike quests, an achievement is very simple to write, but it does require familiarity with a different format of file – achievements themselves are not actually code files, they are just data files. The format is trivial once you understand how it works, but still gives you a considerable degree of flexibility over how your achievements are to be managed.

Unfortunately, achievements are not things that can be integrated into Betterville because of the way they are made active in the achievements handler. All we can do in this chapter is go over the concept. This is then largely a self-contained chapter on how to use our achievements system, with no reference to Betterville. Indeed, that's how the chapter first existed!

Achievements and Quests

So, what is an achievement? For those who are not familiar with these in other games, they may seem just another way of adding quests to the game. This perception is especially muddled by the fact we have had no consistent conceptual architecture when we were developing quests in the past. We have always operated, vaguely, under the guiding principle that a quest should be a puzzle – it is this principle that underlines most of our framing of the rules for quests.

Achievements are rewards for participating and excelling in the game. They are publicly announced (when they are above a certain level), and available for browsing. They have no requirement to be IC – they can be a combination of IC and OOC elements. They can be for achieving certain benchmarks in the game (hitting certain guild level targets), or for passing some threshold of participation (behead 100,000 people as an example)

However, we have numerous quests that aren't actually puzzles in any sense. For example, we have a quest to send a certain number of flowers, another to buy a certain number of town crier messages, and so forth. These would correctly be implemented as achievements (except they predate achievements by Quite Some Time), and it is largely the existence of quests like this that muddy the definitional parameters.

So, let's outline our philosophies on these two gameplay mechanics.

Quests are:

- Puzzles
- Entirely in-character
- Subject to our rules on secrecy and spoilers

Achievements are:

- Participation incentives
- A combination of IC and OOC elements
- In the public domain for discussion

And, let's not forget that, from a development perspective, achievements are tremendously easier to implement than even a simple quest.

How Often Should I Add Achievements?

I would like people to be pretty free in handing out achievements. They are an ideal way of focusing attention on game features that may not have had a lot of publicity, a great way of providing XP to newbies, and an equally good way of rewarding people for playing in ways that do not get proportionately rewarded by our rather combat heavy advancement system.

However...

We don't want to be ridiculous about it. There is a temptation to put in a dozen achievements for each minor development to make sure people pay enough attention. That alas just devalues the rest of them. Imagine someone for example adding a street with five shops, each with a small achievement to go with them where the achievement is 'Buy $X of stuff from this shop'. That's a pretty transparent attempt to curry interest. Instead, a single achievement of 'buy $X of goods from these shops' is more appropriate. Even then it's questionable, since it's not really a **fun** achievement.

I'd say something like this as a rough guideline:

Size of Development	Suggested AP budget
A village or other development of eight/ten rooms	2 AP

A small town (e.g. Lancre Town)	5 AP
A small subsystem	5 AP
A large town (Sto Lat)	10 AP
A moderate subsystem	10 AP
A complex subsystem	15 AP
A small city (DJB)	20 AP
A medium city (Genua)	40 AP
A large city (BP)	60 AP
A huge city (AM)	100 AP

For complex subsystems in a development (the Coffee Nostra, legal systems, etc), you may want to add one or two achievements specifically focused on that subsystem directly, in addition to the area achievements. So you may have 'be arrested for every crime' as an achievement to encourage participation in your subsystem. Lord knows we have enough of them that don't get used enough.

It might be an idea for domains to tot up their existing developments, slot them into the chart above, and budget out the APs accordingly. As an example of Forn back when Genua was put into the game, we'd be looking at a budget of:

Subsystem	AP budget
Genua City	40 AP
Bois	5 AP
Coffee Nostra	15 AP
Legal System	10 AP
Racecourse	5 AP
Wargame	5 AP

Basically, I'm saying that because APs are so easy to give out, we should have a bit of discipline in what we chose to give them for, and not overdo it. Achievements should actually mean something if they are to be worth the name 'achievements'

My First Achievement

Okay, let's start developing our first achievement. To begin with, we need to become familiar with the format of an achievement file (a .ach file). These are located in one of two places:

- /obj/achievements/, for achievements that are mud-wide
- /d/<domain>/achievements/ for achievements that are domain specific.

For achievements within a particular domain, it is the domain administration of that domain who has the authority for approving or rejecting achievements. For /obj/achievements/, you should give me (Drakkos) a nudge. If I am not available, find a fellow creator with sufficient seniority (the rule is – if you have to ask if you have the seniority to approve something, then you don't have it).

Let's take a look at a simple achievement. This one is from the Waterways domain, and is the first achievement that was ever written:

```
::item "waterways:wannabee swashbuckler"::
::->name:: "wannabe swashbuckler"
::->story:: "brought a little bit of Holywood to your actions"
::->level:: ACHIEVEMENT_MINOR
::->criteria:: (["swashes buckled" : 1])
::->category:: ({"sailing"})
::->instructions:: "The very essence of being a swashbuckler is when you do "
  "things that are stupidly theatrical rather than practical.  This "
  "particular achievement is gained by you adhering to that ethos.  You "
  "can work towards this by swinging across to an enemy vessel from the "
  "rigging, fighting on the beams, or sliding down the sail of an enemy "
  "vessel with the help of your favourite stabby weapon."
```

It looks a little bit like a virtual file – it's not though. It's a data file that goes through a handler known as the Data Compiler – this takes this file and converts it into a class file suitable for parsing by the achievements handler.

The first part of the achievement (the 'item' bit) is used internally in the data compiler to build a mapping of achievements in a particular domain. You don't need to worry about this at all, just as long as you're sure that this value does not duplicate anything elsewhere in the system. By convention, we use the domain responsible followed by the name of the achievement for this. This is an internal value – nobody ever sees this.

The rest of the information you set is in the public domain – it'll all be viewable by players in one form or another.

Name is the title your achievement is going to have – this is what will appear when people browse or achieve it.

Story is the little 'jokey' description of what people did to get the achievement. This one, for example, will appear in the player's achievements list as:

```
Wannabe Swashbuckler, in which you brought a little bit of Holywood to your
actions.
```

You do not put in the 'in which you' bit – that's added automatically to provide a measure of consistency with the stories we use for quests.

The level is how much of an achievement this actually is – the values this can be set to are defined in achievements.h. Minor achievements award 5,000 experience points when they are achieved... mythic achievements award a touch over five million. There is quite a gradient from minor to mythic!

We'll come back to the criteria bit – it's the most important part of your achievement file.

Category is, unsurprisingly, what categories this achievement falls under. These are viewable on the website or in the MUD using the 'achievements' command.

Finally, the instructions contain the detailed explanation of the achievement. This is available through the website or through the 'achievements details' command. Our philosophy is that there is no secrecy on achievements, so be as detailed as you can with this.

It doesn't look too intimidating hopefully – as you practise with it, you'll find it's pretty simple to work with.

Criteria

Okay, but what about that criteria part? That looked a bit more complicated!

The criteria is what drives your achievement – it's the bit that tells the achievements handler whether or not a player has passed the threshold you have set for it.

Each player, via the achievements handler, has a multi-purpose mapping of – well, let's call them 'criteria values' that you can manipulate via code. There is no restriction on these – you can call them whatever you like. The criteria of an achievement is set as a mapping of criteria values and the thresholds they must have for the achievement to be granted:

```
::->criteria:: (["swashes buckled" : 1])
```

For this achievement, it is granted when the 'swashes buckled' criteria value is one or greater for a particular player. The achievements handler has no knowledge in advance of what these values are going to be, and it has no idea when this value should be changed – this is where the link between your code and the achievements handler is needed. You tell the achievements handler when it is to change this value, and the handler does the rest.

As an example of this in action, there is a room in the Waterways domains that handles the cool stuff of swinging from ship to ship and sliding down sails with your weapon and such. At the end of any of these processes, they call the following method (if they were successful):

```
void buckle_your_swash (object player) {
  ACHIEVEMENTS_HANDLER->adjust_player_achievement (player->query_name(),
    "swashes buckled", 1);
}
```

So, upon swinging from ship to ship (and doing so sufficiently well), the buckle_your_swash method gets called, and that adjusts the value of 'swashes buckled' for the player by one. That's all that's needed – the handler knows which of these values are related to which criteria, and so it checks each of these in turn to see if the player has met the threshold. If they have, the achievement is awarded.

In addition to the adjust_player_achievement method, there is also a set_player_achievement method that allows you to easily set a specific value for what the achievement should be. For example, you might wish to have some kind of 'accumulator' criteria that requires you to do something correctly a number of times, but to be reset when they fail. There is also a set_highest_player_achievement method that can be used to set a value that can increase, but never fall below the highest value previously attained.

That, in its entirety, defines a simple game achievement. An achievement file (the .ach file) and the code that adjusts the value appropriately. Absolutely everything else is done for you.

Criteria Matching

Criteria are set as a mapping because you can have multiple different values that need to be hit for an achievement to be awarded. For example, we could have the following:

```
::->criteria:: ([
  "swashes buckled" : 1,
  "buckles swashed" : 5,
])
```

With this as a criteria, the player must have the value of one or greater for their swashes buckled value, and five or more for their buckles swashed value.

This may not be exactly what you want – you may wish to provide two or more ways to gain this achievement. The default matching routine for criteria is AND – they get the achievement if they meet all of the criteria values. However, you can also set it to use an OR criteria (they get it if they match **any** of the criteria value) by adding this to your .ach file:

```
::->match_criteria_on:: CRITERIA_OR
```

Alas, you cannot mix and match these (you can't do a compound achievement in which you have to do A and B or C) – if you're getting into that kind of complexity, it's probably becoming more quest-like than anything else.

You may also wish to make use of arrays within your criteria. For example, let's say you wanted an achievement that added a name to an array each time you ate a corpse, and you wanted to grant an achievement when someone had eaten the right four people. You can do this in your criteria:

```
::->criteria:: ([
  "people eaten" : ({"drakkos", "taffyd", "sojan", "wodan"})
])
```

And then when it comes time to adjust the player's achievement values, rather than using adjust_player_achievement or set_player_achievement, you use a pair of methods for manipulating the internal array for the achievement like so:

```
ACHIEVEMENTS_HANDLER->add_to_achievement_array (player->query_name(),
  "people eaten", "drakkos");
```

Arrays in an achievement have a maximum size of ten – this is to keep the memory requirements down and to make sure people don't try to keep an array of every room a player has visited for their 'visit every room on the Disc' achievement. If you start adding to an achievement array that already has ten elements, it will lose the first one and add the new one to the end.

By default, this method will not allow duplicates in an array. If the method call above was called three times, there would still only be an array containing the name "drakkos".

You can force it to have duplicates by adding a last parameter of 1

```
ACHIEVEMENTS_HANDLER->add_to_achievement_array (player->query_name(),
  "people eaten", "drakkos", 1);
```

Call this three times and you end up with an array that contains the name 'drakkos' three times:

```
({"drakkos", "drakkos", "drakkos"})
```

You can also remove things from the array with the remove_from_achievement_array call:

```
ACHIEVEMENTS_HANDLER->remove_from_achievement_array (player->query_name(),
  "people eaten", "drakkos");
```

This will remove one instance of the value, but you can force it remove all instances by adding a 1 to the parameter list;

```
ACHIEVEMENTS_HANDLER->add_to_achievement_array (player->query_name(),
  "people eaten", "drakkos", 1);
```

By default, the handler will attempt to match on array contents – so the player's criteria value has to have in it all of the elements indicated by the criteria. However, you have two other options – you can match on the size of the arrays by adding the following to your achievements file:

```
::->compare_arrays_with:: CRITERIA_SIZEOF
```

Or you can check to see if the player's array contains a specific value:

```
::->compare_arrays_with:: CRITERIA_MEMBER_ARRAY
```

Let's look at how that would work in different situations:

```
::->compare_arrays_with:: CRITERIA_ARRAY_MATCH
::->criteria:: ([
  "people eaten" : ({"drakkos", "taffyd", "sojan", "wodan"})
])
```

This achievement will be granted if and only if the array associated with the 'people eaten' value contains the four specific names mentioned.

```
::->compare_arrays_with:: CRITERIA_SIZEOF
::->criteria:: ([
  "people eaten" : 10
])
```

This achievement will be granted if the people eaten array contains ten elements. It doesn't matter what those elements are.

```
::->compare_arrays_with:: CRITERIA_MEMBER_ARRAY
::->criteria:: ([
  "people eaten" : "drakkos"
])
```

This achievement will be granted if and only if it contains the value 'drakkos'.

What about if we mix it up a bit?

```
::->match_criteria_on: CRITERIA_AND
::->compare_arrays_with:: CRITERIA_MEMBER_ARRAY
::->criteria:: ([
  "people eaten" : ({"drakkos", "taffyd", "sojan", "wodan"})
  "number of people eaten", 100
])
```

This achievement will be granted only if the people eaten array contains the four specific names and the number of people eaten criteria is 100 or higher.

You can create some pretty sophisticated achievements in this way by mixing and matching criteria. Alas, you can only set one way of comparing arrays as of the time of writing – anything more complicated is once again verging into territory best explored by quests.

A Little More Complicated

Ah, you may say – a lot of the achievements in the game are more complicated than that!

This is true, but these are achievements that hook into game handlers or call functions on objects to generate their criteria values. For most achievements, they should absolutely be as simple as indicated above.

However, there is nothing to stop you using a function pointer as your criteria if you want to do something a little more complicated. However, if you do this it needs you to trigger the checking of achievements a little differently.

When you provide a specific criteria value (such as 'swashes buckled'), the achievements handler can work out which of these belong to which achievements. However, if a criteria is a function it has no way of knowing what achievements can belong to that pointer. If the function pointer references an external object (such as a handler), the achievements handler likewise has no way of knowing when the internal state of another object changes. Thus, we need to explicitly tell the achievements handler when the internal state of other objects have been modified.

Note – this_player() in an achievements criteria is a no no. Achievements can be checked while people are not online, or through the web, or in situations where this_player is 0. Likewise, find_player is not to be used in achievement criteria for very similar reasons.

If you wish to check the value of a function defined on a player, you can either use the player handler (which has a number of useful methods), or the method call_function_on_player in the achievements handler – this will ensure the player is online and return the cached value of a criteria if no player can be found. Please, do not attempt to call functions on players directly!

Let's look at a simple example of an achievement using a function pointer:

```
::item "mudlib:stayed the course"::
::->name:: "stayed the course"
::->story:: "stayed the course, you held the line, you kept it all together."
::->level:: ACHIEVEMENT_MINOR
::->criteria:: ([
  (: (time() - load_object(PLAYER_HANDLER)->test_start_time
    ($1->name))      / (60 * 60 * 24) :)  : 180,
])
::->category:: ({"social"})
::->instructions:: "This achievement is awarded upon having a player "
  "account of six months or older."
```

All of this is stuff we've talked about before, with the exception of the criteria:

```
::->criteria:: ([
  (: (time() - load_object(PLAYER_HANDLER)->test_start_time
    ($1->name)) / (60 * 60 * 24) :)  : 180,
])
```

The criteria value here is a function pointer – it gets the current time, then subtracts the player's start time (as queried from the player handler, not the player themselves), and then divides it by the number of seconds in a day. If the value that comes out of that function pointer is greater than or equal to 180, then the achievement is granted.

The first parameter passed to the function pointer is the player details class relating to the player we are checking. The full definition of this class is in achievements.h, but the name parameter of this is usually all you'll need: $1->name.

You may find upon doing this that you get an error relating to achievements not being disambiguated. Don't worry about what this means – you fix it by adding this fairly ugly piece of code;

```
((class player_entry)$1)->name
```

You may only get this error if you are using classes in a header file used by the achievements system.

Anyway, that's an achievement using function pointers. However, the achievements handler has no way of knowing when it should check this, so checking it has to be triggered elsewhere. There are two ways of doing this.

The first is for when achievements belong to a common category and should all be checked at the same time – for example, when you deposit money in the bank it should check all wealth related achievements. For this, we tell the achievements handler to check over all achievements in the wealth category:

```
ACHIEVEMENTS_HANDLER->check_achievement_by_category ("player name",
  ({"wealth"}), 0);
```

The 0 refers to whether or not these are achievements being granted by the legacy command – if it is set to 1, it will suppress all informs except those that go to the player.

If you have a smaller subset of achievements you want to award, you can check specific achievements:

```
ACHIEVEMENTS_HANDLER->check_achievements ("player name", ({"achievements",
  "to", "check"}), 0);
```

These calls go wherever the internal state of the object being checked changes – so for the age ones, they go into the player heart beat (although they don't get triggered every heart beat), for the wealth ones they go wherever the state of a bank account is altered, and so on.

Other Achievement Settings

Titles can go along with achievements, but these (with rare exceptions) should go along with achievements of the LARGE level or higher. Check with your domain administration to see whether or not your achievement can grant a title.

All you have to do to provide a title is to indicate it in your achievement file:

```
::->title:: ({"awesome"})
```

If you want to temporarily switch off an achievement, use the inactive flag. Don't delete the achievement itself, it won't make as much of a difference as you'd like:

```
::->inactive:: 1
```

If you want to set the achievement as first obtained by Great A'Tuin (which we do when we can't know who was the first person to get it), then we do the following:

```
::->first_person:: "Great A'Tuin"
```

If it's in playtesting, then:

```
::->playtester:: 1
```

If you want to give a custom message to the player getting the achievement:

```
::->message:: "You rock!\n"
```

And if you want to register a callback for when an achievement is completed:

```
::->callback:: (: load_object (MY_HANDLER)->register_awesome ($1->name) :)
```

Finally, if you want to register an achievement as being available only to certain guilds:

```
::->available_to_guilds:: ({"warriors", "witches"})
```

Note that this doesn't actually stop non-named guilds from getting the achievement, it just stops it showing up as available for them. You will need to put class restrictions in whatever code you are triggering the achievement from. For example, from the healing ritual code:

```
if (priest->query_guild_ob() == "/std/guilds/priest") {
  ACHIEVEMENTS_HANDLER->adjust_player_achievement
    (this_player()->query_name(), "healing done", diff);
}
```

This ensures that only priests casting the ritual can get the achievement, and it is unavailable to followers using rods or such.

I've written an achievement! What now?

Congratulations!

So, if you're sure you've got permission to make the achievement active, all you do is save your .ach file into the appropriate directory (/obj/achievements/ for mudlib achievements, and the appropriate achievements sub-directory of another domain if it's domain specific).

Before you do anything else, test to make sure the achievement will load:

```
call test_achievement_dir ("the directory of the achievement") /obj/handlers/
achievements_handler
```

If you get an error message doing this, you need to fix your achievement before making it live.

Once you've done this and it compiles correctly, you can just rehash the directory to lodge it in the system.

```
rehash /obj/achievements
```

A second or so later, there will be an announcement to the MUD indicating that a new achievement has entered the game. Then you just sit back and bask in the loving, warming glow of the warming love of our player-base.

Ack, it sucks! How do I get rid of it?

You may have tried to delete the .ach file and found it didn't make much difference – that's because the handler stores details about your achievement so that it can make a note of when it is achieved and how many times and so on. If you are sure you want that achievement to go away, you can do the following call:

```
call clean_achievement ("my achievement") /obj/handlers/achievements_handler
```

And bam, away it will go!

Achievement Levels

It's not necessarily easy to work out what level your achievement should be, because there are all sorts of factors. The rough mapping I use to determine the level is as follows:

Achievement Levels	Points	Notes
Minor	1	Give out like candy. At most, thirty minutes of effort to obtain.
Small	2	An hour or so of effort dedicated to the achievement, or a longer period of time if the achievement is granted by doing what people would do anyway. Breaking 100 street lights might take an hour or so to do and be a small achievement, but getting a common skill to level 150 might take much longer – but it's something you'd be doing anyway
Moderate	3	Half a day or so of effort.
Large	4	A day or two of effort
Huge	5	Three or four days of sustained effort
Massive	6	Several weeks of sustained effort
Epic	7	A year of sustained effort
Legendary	8	Several years of sustained effort
Mythic	9	A decade of sustained effort

Time invested is not a straight-jacket and very difficult to measure. It's just roughly how big an effort you can realistically expect of people when setting your criteria. You also need to factor in things like cost in money, cost in skills, risk, etc, etc. Feel free to nudge me if you want some suggestions on where a particular achievement should sit.

As to the level, feel free to adjust it up or down as needed, although if you're adjusting it two or more levels you should consider why you feel the need to do that. Have a solid reason like 'This is really risky', or 'it costs thousands of dollars', or 'it's moderate even though it takes a year, because it's an achievement you work towards even if you're not paying attention'. For example, login time is something you accrue regardless and is an achievement in the system, as is the time since your first login. Despite the guideline being a decade for mythic, it actually needs 20 years to have passed for 'Dawn of the World' to be achieved because the time passes without you doing anything.

Conclusion

Achievements are a new game-play mechanic for us, and we're learning as we go along where they work and where they don't. From a creating perspective, they excel at adding value to the code that we have with a minimum of creator effort. It's well worth spending a bit of time thinking where you could enhance your domain by drawing people to the right kind of locations through achievements.

So Here We Are Again...

Introduction

I guess it must be fate. We have reached the end of our journey together. There are no more mountains to climb. No more roads to travel. We must simply embrace and wish each other a long, healthy life.

I haven't told you all there is to know about LPC. Oh, far from it. What I have done though, and I hope successfully, is give you the grounding you need to be able to take control of your own learning. Really, that's all I can hope for – that as a result of the intimate candle-lit discussions that we have shared, you have in some way learned, perhaps by osmosis, the skills you need to know where to look for further information.

What's On Your Mind?

Betterville has been quite a complicated development, and it has been tied quite tightly into the discussion of several important programming concepts. It might be worth spending a few moments talking about what you have actually been exposed to in the course of this document.

First of all, we learned about inheritance and how it can be used to build a framework for an easily modified development. We made use of this knowledge throughout our code, making sure we had our own bespoke inherits for every kind of object we used.

Then we talked about data representation, and the uses of the class data-type. Data representation is incredibly important and you should spend a lot of time thinking about how you approach this in any even moderately complex development.

We introduced the topic of add_commands, and the various syntaxes and patterns that go with it. This opens up a world of interaction possibilities within our rooms and objects. This allowed us to develop our first quest, and that in turn led to our discussion of the quest handler and the library handler, as well as the quest utility inherits that can be used to simplify data management.

Inherits are one half of a code architecture – the other half comes from handlers, and we built no fewer than two of these in the process of developing Betterville.

We talked about events, and how we can trap these to provide responsive behaviour in our NPCs, and how data persistence can be implemented on the MUD. We also talked about the smart fighter code and how you can make your NPCs throw some interesting attacks out there when they find themselves in combat.

As part of our shop, we made an item that had to save its state between logins, and that lead us to the discussion about the auto loading system. We wrote a couple of spells, and we wrote an effect. We learned about achievements, and we talked about function pointers. We've talked about an awful lot in this material, and all of it is important for you to know as a fully rounded Discworld creator.

This does not complete your toolbox – there's all sorts of things we haven't talked about. We haven't talked about terrains, or socket programming, or coding for the web - you're well placed now to learn about all of that stuff yourself. There is always more to learn, but now you will not only know enough to code some very interesting objects, you'll also have the necessary background information to be able to understand new and interesting code that comes your way.

Help Us To Help You

LPC is a niche language, and it's not especially well documented. The same thing can be said for large portions of our Mudlib. Here be dragons, as they say. There are all sorts of clever features and little tricks that people know how to do, and every now and again you'll find one that throws you. There's nothing I can do to get around that, except perhaps spend the next ten years of my life writing more and more of this kind of thing. I don't intend to do that.

However, there's no reason that you can't help yourself and the people around you as you make your way through the world of being a Discworld creator. Whenever you discover an undocumented feature, then document it. When you've spent a couple of hours puzzling over a bizarre bug, then tell people about it. We're all in this together – don't think that your knowledge is in any way commonplace. You may be the only one who knows what you know, or the only one who spent time working out how that weird situation manifested itself.

We have a wiki that is a great place to record little essays on things that you've discovered, and there is always the learning board, or the Discworld Oracle. More than anything else, if you're doing something strange or complicated, then document it! Better still, do it in a less complicated way. Very few creators are still around from the very first days of the MUD. Along the way, we lost an awful lot of knowledge – every day we live with the consequences of that in the shape of design decisions that were made, or strange coding syntaxes that were implemented.

In ten years time, the MUD will hopefully still be here – you may not be. Part of your legacy should be what you have contributed to the knowledge-base that we collaboratively build.

Where Do You Go From Here?

Now that you've finished with this material, where do you go? What should you be doing next to move on to the next level of being a Discworld creator?

I am a big believer that the only way you get better is to set yourself a task that you don't know how to complete. Don't stay inside your comfort zone – try to find something new and exciting in every project you work with. There is very little that cannot be achieved within the Discworld Mudlib and the framework that LPC provides, and the struggle to do things that you have never done before is where your real learning begins. You don't learn how to code from a book, you learn by sitting down and puzzling over a difficult problem.

You've got access to a lot of code as a Discworld creator. Perhaps not access to change it, but you can certainly read code without any difficulty. Reading the huge amount of code we have in the various domains and Mudlib directories will introduce you to new and quirky (and not so quirky) ways of building complex objects. From this point on though, it's all on you.

While I have no intention of writing an LPC for Dummies 3, there are several more general resources I can recommend for those who want to become better coders. The language you use is just a frame on which you hang your development – all programming is essentially the same thing, at least within particular families of code. Come speak to me if you have specific requests for things you'd like to learn more about.

Conclusion

So long, farewell, adieu. This text has been a long time coming. LPC for Dummies one, the first edition, was written in October of the year 2000. I'd been a creator for about a year and a half. People liked it though, and I was constantly asked about a sequel that would address the more complicated parts of LPC development. I've been promising it ever since then, and I am delighted to finally be able to provide it.

My plans for LPC for Dummies 2 changed a lot over the years. Really now it's one of four tightly related works. Together they form a kind of informal ‘Discworld Creator Manual'. A lot of material that was originally planned for LPC for Dummies 2 made its way into other works, and you should consider the these texts to be four separate sections of one larger book. I hope, in your travels, you find it of use to you.

Reader Exercises

Introduction

There are a few bits and pieces that have been left out of the main text so as to provide a pretext for you to add them yourselves. None of these require you to do anything that you didn't learn about during the course of LPC For Dummies one and two. Full answers to these exercises are available.

Some of these are easier than others, but all of them involve a degree of practicing for yourself what we have worked through in our example development.

Exercises

Ascending the Library

One of the things we decided in Being A Better Creator was that ascending through the levels of the library should give a reward to those who reach the top. That reward would be to have their name registered in the library below.

For this, you'll either need to create a new handler (a good option) or repurpose an existing handler (not so good option), or hook into the quest library (a good, but limiting option).

Modify the code so that players who meet the beast are given immortality in the library.

Researching Players in Library

We indentified one of the ways in which socialisers would be drawn to our Betterville library is if they were given an opportunity to research other players there.

1. Decide upon a suitable form of information that researching a player should yield
2. Implement that into the research command that already exists.

Note that you will have to incorporate this functionality in such a way that it doesn't impact on the existing quest hint system.

Researching Other Concepts

Our library should also have information about concepts, so as to provide a narrative framework. In addition to allowing research of wizards and players, make it so players can also research concepts such as ‘fairytales’, ‘genua’, and all of the other varied things we might want to add a little story for.

Beast disemboweling

We have a handler in place for dealing with tracking players to be disemboweled by the beast, but he doesn’t actually do it. Modify the code so that he checks for people who need to be disemboweled when they come into contact with him.

Once you have done that, have him enter combat with transgressors, and add a combat action that disembowels a player if they fail some kind of skill-check.

Rubble red herring

The secret passage on the second floor was supposed to be a hidden quest – the way up to the next floor was supposed to be blocked by rubble. Because we are Bastards, we were also going to make these a red herring.

Add in the necessary descriptive text here, and add some misleading add_command functions to let players ‘manipulate’ the rubble without ever letting them ascend to the next level.

Lady Tempesre

Lady Tempesre is supposed to be fulfilling the part of a character in a kind of mini-fairytale. Alas, she has no clothes to make that take complete. Dress her up in a suitable ‘evil witch’ outfit. Give her some other.perception at the same time.

Also, we never did dress the Dozy Girls, so finish off the stock for the shop and incorporate it randomly yet appropriately into the setup of the girls.

Library Index Card

The library should have an index card system that shows what things a player can research in it. Write the code that does that – it should list all of the wizards, all of the players who are present, and all of the other concepts that should be represented.

Research Skill check

It’s not easy to research in a library – certainly not something you always do and always get right. Incorporate skill-checks into the library so that sometimes you don’t find anything, and sometimes you find misleading information.

Helpfiles for library

The library has a semi-complex syntax to go with it. We don't want our quests to be 'guess the syntax' quests, so create a help-file for the library that details the syntax and functionality available.

Add it to the library so that 'help here' functions correctly.

Resetting The State

When a player gets horribly, terribly confused, they may wish to simply give up and return the library back to its initial state. Give them a command that does this. Make sure the command fails if they have already completed the quest, or if the library is already correctly sorted.

Send Suggestions

Do you have ideas for exercises that would be cool, or are you trying to do something new in Betterville and just can't make it work? Send them to me, Drakkos, and we can look at incorporating them into this section!