LPC For Dummies

Book Two

by Michael Heron (Drakkos) and the Discworld MUD creator team

Table of Contents

1. The Mudlib Strikes Back

1.1. Introduction

Welcome to *LPC for Dummies 2* — the intermediate level material for Discworld creators. Before working through this volume, you should be sure that you understand the material presented in *LPC For Dummies 1* and *Being A Better Creator*, because I'm going to assume that you've thoroughly read and understood both of them. This is especially important for the coding side of things — seriously, if you don't understand every single thing in *LPC For Dummies 1*, you shouldn't even think about attempting *LPC For Dummies 2*. I'm going to expect a lot from you throughout this volume.

Once we progress beyond the basics of LPC coding, it becomes possible to do some very fun things. However, the theory and practice both get 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 what we're going to be doing as our worked example.

1.2. Betterville

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

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

In particular, we've 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're also going to have a range of NPCs to populate our area. First we have the Young Romantics — 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 backstory. High interactivity is important here.

Finally, we've 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 our gold-digging young romantics.

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" — you wouldn't learn a lot by doing a dozen variations on the same theme, and so we'll talk about how to do something once and leave it as an exercise for the reader to fill in the blanks.

1.3. The Village Plan

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

```
    6
   / \
  5 + 7
   \ /
    4
   /
  3
  |
  2
 / \
1   8
     \
      9
```

Our first step is going to be to set up this framework. We're going to do it a little bit differently from how we did Learnville. This is *LPC For Dummies 2,* after all — we do things Bigger and Better here!

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.

1.4. 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!

1.5. 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!

2. Inheritance

2.1. 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 area-level functionality. In this chapter, we're going to build a series of custom inherits for our new and improved area. These will be specialisations of the existing inherits (we're not going to need to put 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.

2.2. 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 instead 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, but in brief: what it's doing is making sure that `setup()` and `reset()` get called once and once only, and at the right time, when an object is created; without this, you get weird things like doubled-up add_items and such.*

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 a specific `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();
  }
}
```

This instantly 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 rather than `/std/outside`.

It's worth putting a bit of thought into where your inherits will live within your development's directory structure. One option is to store them all in a single directory. This would be appropriate if many or all of your inherits are used in several different places in your development, as it makes it totally clear where to find them.

Another option is to keep them as close as possible to the code to which they relate. This could be appropriate if the primary reason for having an inherit is to avoid copy-pasting descriptions between several files in the same subdirectory; for example, if you were writing an area with several streets, each of which had its own ambience, you might wish to keep `main_street_inherit.c` in your `/main_street/` subdirectory along with `main_street_01.c`, `main_street_02.c`, etc.

* There can be lots of create() methods as you work your way up an inheritance tree, and every create() needs to call the one that sits above it before it does its own thing. So we have an integer variable called do_setup, and we add 1 to it every time one of those earlier create() methods is called and remove 1 from it every time the earlier create() finishes. Then checking for !do_setup (i.e. checking that do_setup == 0) means that you are checking for the point at which all the create methods have run and you're safe to move on to doing setup() and reset().

You'll see both these strategies used throughout the MUD, but for the sake of convention, all the inherits we talk about in this book will be stored in the `/inherits/` subdirectory of our `betterville` folder. This one will be stored as `outside_room.c`.

2.3. Hooking It All Up

Our next step is to make this inherit freely available in our project, so we set up `path.h`:

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

Now that we have this, let's put the architecture of our new village in place. We do this the same way as we did for Learnville, except we have a new and exciting inherit to 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. Call these rooms `betterville_01.c`, `betterville_02.c`, ..., `betterville_09.c`, and put them in your /rooms/ subdirectory. You should remember how to add the exits from *LPC For Dummies 1* in the chapter entitled “My First Area”, but 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
You better believe it!
```

Now we are well placed to put in whatever functionality we want in our inherit. We can add move zones, 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!

One caveat: when working with your own inherits, updating 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 if you can avoid it, though, because you'll end up updating a lot more objects than you need to. If you're going to do it, check "help dupdate" and make sure to use the optional <depth object> parameter. The whenl command can be helpful in checking whether your inherit tree is up to date.

```
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();
  }
}
```

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();
  }
}
```

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

2.4. 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? For example, if we wanted to write a complex piece of code to be attached to a search function that would work in all our village rooms, we wouldn't want to copy and paste it into both inherits. 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() {
}
```

Yes, that's the entire file right now. Notice our `create` method isn't actually doing anything at the moment, but this will change later on, I promise. We're just getting the structure set up here so we know what's what. We'll save this shared inherit as `betterville_room.c`, and then make use of it within `outside_room.c`:

```
#include "path.h"

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

and `inside_room.c`:

```
#include "path.h"

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

2.5. 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.c`:

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

And in inside_room.c:

```
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 create or the other, then we could do that too by omitting a call to the relevant (or, rather, the irrelevant) parent.

2.6. Our Betterville Room Inherit

We can now start thinking about code that we might want to move from outside_room.c and inside_room.c into betterville_room.c — to start with, we have two lines that appear in both of those inherits that might as well go into their parent:

```
  set_light( 100 );
  add_zone( "betterville" );
```

Now, the problem we have here is that since our betterville_room.c inherit doesn't inherit any of the room code, we can't use any of the normal room functions like add_item, set_light, and add_zone. Or, rather, we can, but we need to do it less succinctly. If we do the below, we will get an error when we try an update:

```
void create() {
  set_light( 100 );
  add_zone( "betterville" );
}
```

However, we can do something like this instead:

```
void create() {
  this_object()->set_light( 100 );
  this_object()->add_zone( "betterville" );
}
```

Using this_object() allows us to treat our code as a unified whole — there is no add_zone function defined in betterville_room.c or in anything it inherits (because it doesn't inherit anything) but there *is* a definition of this function in the "package" of betterville_room.c and inside_room.c‡. Using this_object() lets us call methods on the whole package rather than just on the constituent bits.

‡ Because inside_room.c inherits /std/room/basic_room.c, which defines the set_zone function. You can confirm this with the find command, e.g. find set_zone inside_room.c

And now we don’t need to copy-paste those lines between our inside and outside inherits. So, `outside_room.c` becomes:

```
#include "path.h"

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

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

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

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

And `inside_room.c` becomes:

```
#include "path.h"

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

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

  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();
  }
}
```

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.c`. If they’re for inside rooms only, they go in `inside_room.c`. If they should be shared between both, I can put the code in `betterville_room.c`.

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 — it’s not an insoluble problem by any means.

2.7. 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 my inherit 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!\n" );
}
```

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

```
inherit INHERITS + "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*. Setting a variable to be *private* 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 if they want to, but changing that won't change the state of the one in the inherit. This forces people to go through our `adjust_balance` method. If we want them to be able to directly set and/or get the value of the variable, we can also supply *accessor methods*, which come in two types: *setters* and *getters*. These allow the variable to be manipulated in a way that we have control over:

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

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

You should only supply setters and getters for private variables if you want other code to be able to work with them. You don't always have to supply both; for example, you might want people to be able to find out the value of the variable but not to alter it, in which case you would supply a getter but no setter.§

2.8. The Impact Of Change

If we have a public variable in our inherit, we have to assume that someone else is making use of that variable in their own objects, and if we change it we will break their code. That's bad karma! We limit variables to being private to keep our options open — if I want to change a variable from an int to a float, then I can't easily do that if it's set as 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. Methods can also be set to private, and 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!\n" );
}
```

§ Note that when LPC returns a non-scalar variable (e.g. an array, mapping, object, etc) from a function, it does this *by reference*, which means that the caller can modify the contents. For example, if a handler has a global variable `private string *_names = ({ "drakkos", "kake" });` and a getter function `string *get_names() { return _names; }`, then some entirely separate object can do `mynames = HANDLER->get_names(); mynames[1] = "fran";` and this will change the actual array in the original handler so that anyone else who calls `get_names()` will get `"fran"` instead of `"kake"`! You can avoid this by using `copy()`, e.g. `string *get_names { return copy( _names ); }`.

Sometimes, though, we don't want to be quite this restrictive. Maybe we want to give access to people who are making use of our inherit directly, 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.

2.9. 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!

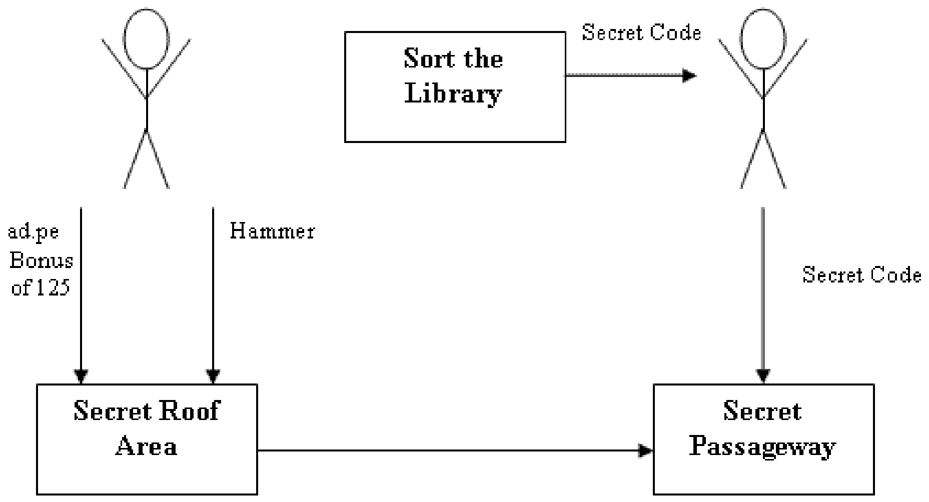
3. The Library Quest

3.1. 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 us to put a dynamic framework around it so that it can't be solved simply by following a set of instructions. 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!

3.2. 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 to 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, everyday 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.
  - We need to be able to get a random list of these.
- We need to store some category headers.
  - We need to be able to get the list of these.
- We need to store how a player has sorted books.
  - We need to know which category a player has associated with each book.
  - We need to let players move books from category to category.
  - We need to let players reset their entire categorisation.
  - We need to let players add books to categories.
  - 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, _book2, _book3, _book4, _category1, _category2, _category3,
  _category4;
mapping _books_and_categories;

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, *_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 Neverending 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.

3.3. 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. A class in LPC is 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 now designate things as being of type `class book`:

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

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

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

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

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

```
class book new_book = 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 *_all_books = ({ });

void add_book( string book_title, string book_author, string book_blurb,
               string *book_categories ) {
  class book new_book = new( class book,
    title: book_title,
    author: book_author,
    blurb: book_blurb,
    categories: book_categories
  );

  _all_books += ({ new_book });
}

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

  foreach ( class book this_book in _all_books ) {
    foreach ( string c in this_book->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 know the title of a book, and I want to find out its blurb? First, I need a way of finding that book element in the array, so I'll write a function to do that:

```
int find_book( string title ) {
  for ( int i = 0; i < sizeof( _all_books ); i++ ) {
    if ( _all_books[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 _all_books[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.

3.4. 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 as information about the player, or do we store it as information about the room?

If we store it as information about the player, then multiple players will be able to work in the same library without their actions impacting on the state of the other players. If we store it as information 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 library "instance", let's just decide we're going to store the state of the room — so when one player sorts a book on a shelf, everyone else in the room sees that book as being on that shelf.

We will need some more variables in our room to store which books are where. This works pretty simply as a mapping — book title X is on bookshelf Y:[**]

[**] If you're wondering why we return -1 instead of 0 when we can't find a book, the reason is that arrays start indexing at 0, so if we returned 0 for "cannot find book", then that would be indistinguishable from "found book, it's the first entry in the _all_books array". It's the same reason why `member_array()` returns -1 for "cannot find item". We often use -1 as a flag for "couldn't find the thing" or "something went wrong" in situations where a return value of 0 *doesn't* indicate not-found/gone-wrong.

```
mapping _sorted;

void assign_book_to_shelf( string title, string category ) {
  // The calling code is going to be responsible for giving appropriate
  // failure messages if the book or category don't exist, but we check here
  // too just to make absolutely sure we don't store bad data.
  int i = find_book( title );

  if ( i == -1 || ( member_array( category, query_categories() ) == -1 ) {
    return;
  }

  // All OK, so put it on the chosen bookshelf.
  _sorted[title] = category;
}
```

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

3.5. 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 a clue of what their 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 new_book;
  string *valid_blurbs;
  string *cats = ({ "romance", "action", "childrens" });
  string my_blurb, my_cat;

  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 carriage 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 );

  new_book = new( class book,
    title: book_title,
    author: book_author,
    blurb: my_blurb,
    categories: ({ my_cat })
  );

  _all_books += ({ new_book });
}
```

Now we have a quest architecture in place — we give it 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.

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.

3.6. 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 = ({ });

  foreach ( class this_book in _all_books ) {
    if ( !_sorted[this_book->title] ) {
      unsorted += ({ this_book->title });
    }
  }

  return unsorted;
}
```

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

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

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

```
string reset_sorted() {
  _sorted = ([ ]);
}
```

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 sanitised manner. For example, we may give the player a command in a room:

```
sort <book title> into category <category>
```

When they use this command, it hooks into the functions we've written — specifically, it will call the `assign_book_to_shelf()` function, providing the book title and category they give as arguments. We need to then give meaningful output to the player indicating what has happened. That's for another chapter though.

3.7. Conclusion

This chapter has introduced you to a new, powerful data type: the class. I am a big fan of classes, as 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.

4. Adding Commands

4.1. 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 window-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.

4.2. Deciding On A User Interface

There are certain elements of building a user interface that are universal rules to which we must adhere. You might find code already in the game that violates these rules, but all new code should obey them.

A good user interface has the following traits:

- It is consistent.
- It is predictable.
- It allows users to undo mistakes.
- It 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. Violations of this principle 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 tiny yak-like creatures 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 the sound of shaving 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, you are effectively telling the player that anything they think they have learned may be discarded at any time. The user then has no way of making meaningful decisions about how to interact 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 your stats 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 to start over.

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

```
Try again.
```

How do I 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: `add_command`.

[††] If you don't yet know what yak-shaving means in the context of programming, count yourself lucky.

4.3. Adding Commands

You've already added commands in a simple way in *LPC for Dummies 1*, where we added them 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. When a player types in a string such as `stab drakkos with knife`, the parser breaks that up into the command (`stab`), the objects involved (`drakkos` and `knife`), and the rest of the text (which is just there to make 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 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 (i.e. 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
```

So `args` 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`:

‡‡ If you think it looks a bit messy to have all those parameters passed to the function when you don't actually care about most of them, don't worry! We'll discuss a way of tidying it up in section 15.5.

```
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. The person using the command will see:

```
You test the command.
```

Other people will 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 value of `1` means that the command succeeded and that a success message (added by us) should be displayed. A return value of `0` means that the command failed and a failure 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 failure message goes only to the player, and should be used when the actual functionality of a command has failed to even get started — 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 might 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.

These ways of invoking add_succeeded_mess and add_failed_mess are fairly simplistic, but we can make them more powerful; for example, there are a bunch of other `$` codes we can use. We’ll talk about that a little later in this chapter.

4.4. 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. Here's a list of codes you can use in your command pattern, and the things they are used for:

Code	What it matches
`<indirect>`	An object.
`<direct>`	The object in which the function associated with a command is defined.
`<string>`	At least one word, and possibly more.
`<word>`	A single word.
`<number>`	An integer expressed in digits, such as 3 or -42.
`<decimal>`	A number with a decimal point in it, such as -1.5 or 3.142.
`<ordinal>`	An ordinal number such as "third" or "42nd".
`<fraction>`	A fraction, such as 3/4, 5/6, etc.
`<preposition>`[§§]	Such as "to", "from", etc.

We can also make these codes more specific in terms of what they will match, by adding options separated with colons. We'll focus on objects here, but you can also do this for 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	What it matches
`:object`	Any object in the player[†††] or the room.
`:living`	A living object in the player or the room.
`:distant-living`	A living object found via the function `find_living`.
`:any-living`	Matches living objects in the room first, and then distant objects later. The strings `"someone"`, `"everyone"` and `"creators"` are also valid matches .
`:player`	Only matches a player.

§§ We don't seem to use this one very much on the MUD — possibly because it isn't at all clear to players which words are valid here, which violates the principle of clarity. So I recommend that you don't use it either.
*** See `help add_command_details` for information on the options for strings and numbers (and also for a more in-depth look at objects, since we are providing more of a summary here)
††† We're using "player" here to mean the living object that is using the command. This might be a player, but NPCs can use commands too. We're just using "player" to simplify the explanation.

`:wiz-present`	This one does complex and fancy stuff if the person using the command is a creator. See `help wiz-present` for details.

We can combine these with our previous set of codes as we choose; for example:

```
<indirect:living>
```

or

```
<direct:object>
```

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

Code	**Where it looks for a match**
`:me`	The inventory of the player performing the command.
`:here`	The inventory of the environment of the player.
`:here-me`	The environment first, and then the inventory of the player.
`:me-here`	The inventory of the player, 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 inside the single quotation marks is what will be displayed to the user instead of the pattern itself.

It's important to remember that when it comes to indirect objects, `here` doesn't just include things that are *directly* inside the enviroment (e.g. things lying on the floor, people standing in the room) — it also includes things inside those things, as long as the player can reference them directly. So if there's a cup on a table, or a rather nice bracelet being worn by another person standing here, then because I can `look cup on table` and `look bracelet on drakkos`, the cup and bracelet will *also* be matched by `object:here`.[‡‡‡]

This means that if you have a command that uses `<indirect>`, and you want to stop your command from being usable on things in other people's inventories, you *must* check the environment of any objects you match before you do to them whatever it is your command does to things.

4.5. 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.

4.6. Direct and Indirect Objects

The different between the direct and indirect objects 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.

‡‡‡ It generally stops there, and so, for example, since you can't `look bracelet in backpack in drakkos`, Drakkos' stashed-away bracelet is safe from badly-coded commands. (Those of you with experience of thievery will now be wondering how the `rifle` command manages to work — the answer is that it doesn't use object matching, but instead uses string matching and figures out for itself what the thief is trying to nab.)

You could use it productively in the meantime to make it clear which object you are referring to. If you want your new pair of extra-armoured boots to provide a `kick` command, you would add the command like so:

```
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 (e.g. `"kick drakkos with boots except damaged boots"`) and gives you the specific boots object that the player has chosen to use, so you can make sure, for example, that the player is actually wearing them.

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

4.7. 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 match 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.

4.8. Tannah's Pattern Matcher

Adding commands and working out what the parameters are is horrendously complicated. Luckily, Our 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 at `/d/learning/items/matcher.c` (or just do `req learning 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!

4.9. Syntax Details

We mentioned above that the pattern of our command is used to build the information that players will see when they type `syntax <command>`. However, the command examples we've looked at so far are missing an important piece of information — the *syntax details*. The syntax details are the little explanation that shows up to tell you more about what a command does. For example:

```
> syntax syntax
Forms of syntax available for the command "syntax":
syntax <verb>                         Show the syntaxes for a particular
                                      command.
syntax <verb> {brief|verbose}         Show brief or verbose syntaxes for
                                      a particular command.
```

If we want to supply these for our own commands (and we should!) we need to use a slightly longer way of invoking `add_command`. Instead of this:

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

we do this:

```
void init() {
  ::init();
  add_command( "test", "<string>", 0, "Test something out." );
}
```

As before, our first argument is the command name and the second is the pattern, but we have two extra arguments here. The third argument defines the function that will handle what happens when a player uses this command, and we're setting it to `0` to tell the MUD to just figure it out from the command name; in this case, it will call `do_test()`[§§§]. And then the fourth argument is our syntax details.

4.10. Conclusion

We've now been introducted to `add_command`, a 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've mastered it, you'll be able to add complex, interesting functionality to everything you create.

§§§ If we want to supply syntax details, they always need to be the fourth argument, which means we have to have a third argument, since otherwise the fourth argument would be the third argument. The third argument can be non-zero, and in fact it often is; we'll discuss that further in section 15.5.

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.

5. The Library Room

5.1. 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.

5.2. 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 *_all_books = ({ });
mapping _sorted;

void reset_sorted();
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" );
  add_exit( "west", ROOMS + "betterville_09", "door" );
  setup_quest();
  reset_sorted();
}

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

  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 carriage 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 );
  new_book = new( class book,
    title: book_title,
    author: book_author,
    blurb: my_blurb,
    categories: ({ my_cat })
  );

  _all_books += ({ new_book });
}

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

  foreach ( class book this_book in _all_books ) {
    foreach ( string c in this_book->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( _all_books ); i++ ) {
    if ( _all_books[i]->title == title ) {
      return i;
    }
  }

  return -1;
}

void assign_book_to_shelf( string title, string category ) {
  // The calling code is going to be responsible for giving appropriate
  // failure messages if the book or category don't exist, but we check here
  // too just to make absolutely sure we don't store bad data.
  int i = find_book( title );

  if ( i == -1 || ( member_array( category, query_categories() ) == -1 ) {
    return;
  }

  // All OK, so put it on the chosen bookshelf.
  _sorted[title] = category;
}

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

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

string *query_unsorted_books() {
  string *unsorted = ({ });

  foreach ( class book this_book in _all_books ) {
    if ( !_sorted[this_book->title] ) {
      unsorted += ({ this_book->title });
    }
  }

  return unsorted;
}

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

void reset_sorted() {
  _sorted = ([ ]);
}
```

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 the information they need through interacting with the room. First of all, they need to know which books need to be sorted, so let's tie that into an add_item.

Because the description of the add_item will have to be dynamic, we need to use a *function pointer.* We'll talk about these in more detail in chapter 15, but for now, all you need to know is that when an add_item's description is something of the form `(: my_function :)`, then when someone looks at it, what they will see is whatever is returned by the function `my_function()`. The `(: :)` notation is how we tell the MUD that we want to use a function pointer.

The skeleton for our code is 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" );

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

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

```
> l books
blah
```

That's a bit blah, so now we need to build our `books_to_sort()` function. It should have the following behaviour:

- If there are books that haven't been shelved, we should list those books.
- If there are no books left to shelve, it should indicate this 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( ({ "book shelf", "bookshelf" }), (: books_on_shelves :) );
```

We now need to implement the `books_on_shelves()` function. Our `_sorted` 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 ( !sizeof( _sorted ) ) {
    return "No books have been sorted onto the shelves.";
  } else {
    foreach ( string title, string category in _sorted ) {
      ret += "The book \"" + title
        + "\" 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 (a) build that into the bookshelves, or (b) provide a cue to look at another add_item. I favour option (b), firstly because this is static rather than dynamic information, and secondly because if we pile all our information into one add_item then we risk overloading the player. So we change our bookshelf add_item a little, to point to an add_item for some labels that are conveniently stuck to each shelf:

```
string books_on_shelves() {
  string ret = "The bookshelves are arranged according to categories.  "
    "Rather tattered-looking labels indicate which category belongs to "
    "which shelf.";

  if ( !sizeof( _sorted ) ) {
    ret += "No books have been sorted onto the shelves.";
  } else {
    foreach ( string title, string category in _sorted ) {
      ret += "The book \"" + title
        + "\" 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:

```
[...]
  add_item( "tattered-looking label", (: book_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.

5.3. Guess The Syntax Quests

I am as guilty of this as anyone, but we must be mindful when coding an add_command to make sure 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 realising 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 for 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 should make sure that the instructions you write for the quest page on the MUD website are good and clear; you might also consider making the instructions available in a help file attached to the room, objects or NPCs involved in a quest. The logical process you go through to solve the quest should be the task, not finding the actual commands you need to use.

5.4. 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, it’s very likely that player who’re looking for more information about a book will try to “read” it, 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: "
      + query_multiple_short( unsorted ) + ".  They are just begging to "
      "be given a good sorting.  Each has a blurb on the back that "
      "might help in discovering which category they belong to.";
  }
}
```

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

```
void init() {
  ::init();
  add_command( "read", "blurb on <string'book'>", 0, "Read the blurb on "
    "the back of a book." );
}
```

And then we add the code that handles the functionality. It's not complex, since 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 title = args[0];
  string blurb = query_book_blurb( title );

  if ( !blurb ) {
    add_failed_mess( "That book doesn't 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;
}
```

When the player tries this, they will see something along these lines:

```
> read blurb on Clarabelle
The blurb on the book reads: "A story about a lost cow, for children."
You pick up a book and read the back.
```

Hm, that's not quite right, though — it's telling you what the blurb reads *before* it's telling you that you read the blurb. You may recall that we had a similar issue in *LPC For Dummies 1*, where we saw what the magic hate ball was telling us before we saw ourselves shaking the ball, and I promised you then that *LPC For Dummies 2* would explain how to fix it. We have now reached the point where it can be explained! Aren't you glad you carried on reading these texts? I certainly am.

The key to the solution is knowing that `add_succeeded_mess()` can be called in a different way — with an array of two strings as its argument instead of just a string. The first string in the array is shown to the person who has successfully used the command, and the second string is shown to anyone else in the room. So we can replace the above with:

```
int do_read( object *indirect_obs, string dir_match, string indir_match,
             mixed *args, string pattern ) {
  string title = args[0];
  string blurb = query_book_blurb( title );

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

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

And now we see:

```
> read blurb on Clarabelle
You pick up a book and read the back.  The blurb on the book reads: "A story
about a lost cow, for children."
```

With that, the blurb has been dealt with. Except, not entirely — it doesn't take into account the language the blurb is written in, and the player's skill in reading that language. We'll come back to that later.

5.5. 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'>",
    0, "Sort a specific book into a specific category." );
  add_command( "read", "blurb on <string'book'>", 0, "Read the blurb on "
    "the back of a book." );
}

int do_sort( object *indirect_obs, string dir_match, string indir_match,
             mixed *args, string pattern ) {
  string title = 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 Clarabelle into category childrens
```

Or just:

```
> sort Clarabelle into childrens
```

The syntax message should hopefully be self-explanatory:

```
> syntax sort
Forms of syntax available for the command "sort":
sort <book> into [category] <category>  Sort a specific book into a specific
                                        category.
```

No need to guess the syntax, it's all there wrapped up in the command. That's good practice — 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 trying to sort one 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 title = args[0];
  string category = args[1];

  if ( find_book( title ) == -1 ) {
    add_failed_mess( "That book doesn't seem to exist.\n" );
    return 0;
  } else if ( member_array( category, query_categories() ) == -1 ) {
    add_failed_mess( "That category doesn't seem to exist.\n" );
    return 0;
  }

  return 1;
}
```

Finally, if it turns out that they're 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 title = args[0];
  string category = args[1];

  if ( find_book( title ) == -1 ) {
    add_failed_mess( "That book doesn't seem to exist.\n" );
    return 0;
  } else if ( member_array( category, query_categories() ) == -1 ) {
    add_failed_mess( "That category doesn't seem to exist.\n" );
    return 0;
  }

  assign_book_to_shelf( title, category );

  add_succeeded_mess( "$N sort$s the book \"" + title
    + "\" 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 title, string category in _sorted ) {
    i = find_book( title );

    if ( i == -1
         || member_array( category, _all_books[i]->categories ) == -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 title = 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;
  } else if ( find_book( title ) == -1 ) {
    add_failed_mess( "That book doesn't seem to exist.\n" );
    return 0;
  } else if ( member_array( category, query_categories() ) == -1 ) {
    add_failed_mess( "That category doesn't seem to exist.\n" );
    return 0;
  }

  assign_book_to_shelf( title, 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 \"" + title
    + "\" into category \""  + category + "\".\n" );
  return 1;
}
```

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

5.6. 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:

```
> sort clarabelle into category childrens
That book doesn't seem to exist.
```

Oh no! The reason for this is that we have capitalisation 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 `title_case()` throughout our code; specifically, we’re going to store the book titles as lowercase and then capitalise them as they’re shown to the user.**** So when we add a book, we’ll lowercase it like so:

**** Note that we can get away with this because we have complete control over the book titles, and hence we can make sure we don’t have books called “The Adventures of PHLEGM” or “Nobby of the d’Nobbevilles”, which `title_case()` would capitalise wrongly. If we had to make this work for arbitrary titles, we’d have to store both the canonical capitalisation (for display) and the `lower_case()` version (for comparisons).

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

Now we have no need to concern ourselves about the capitalisation of user input, because once we've processed it, it's all be 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 title, string category in _sorted ) {
    ret += "The book \"" + title_case( title )
      + "\" has been put on the shelf for the category \""
      + title_case( category ) + "\".\n";
  }
}
```

And in our `do_sort()`:

```
add_succeeded_mess( "$N sort$s the book \"" + title_case( title )
  + "\" into category \"" + title_case( 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 ) {
  string *new_arr = ({ });

  foreach( string str in arr ) {
    new_arr += ({ "\"" + title_case( str ) + "\"" });
  }

  return query_multiple_short( new_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() ) + ".";
}
```

†††† Note that we create a new array in this function rather than modifying the entries of the existing array within the loop. The reason for this is that modifying a data structure while you're iterating over it is generally a bad idea. It would be OK in this instance, but it's best not to get into the habit of it, since it can cause some hard-to-find bugs.

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:

```
string my_npc_chat = "$L$[morporkian]This is in Morporkian.$L$\n";
```

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

```
string my_text = "$L$[read:morporkian]This is in Morporkian.$L$\n";
```

So therein lies our solution:

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

  add_succeeded_mess( ({
   "You pick up a book and read the back.  The blurb on the book reads: \""
      "$L$[read:morporkian]" + blurb + "$L$\".\n",
   "$N pick$s up a book and read$s the back.\n",
  }) );

   ...
}
```

And then Bob is the sibling of your parent!

5.7. 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!

6. Quest Handlers

6.1. 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.

6.2. 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*‡‡‡‡, which holds the records of a specific player with regard 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 — when you come to write a quest for the live game, it's the leader of the relevant domain who will have to add the quest and make it available to players. There's a web-interface you can explore at https://discworld.starturtle.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 the library handler available 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" );
}
```

‡‡‡‡ The handler known as the "library" should not be confused with our specific library in this specific village. It's a separate thing used for all quests.

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.

6.3. 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 by calling the method `set_player_quest_info()` 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( 10000 );
  return sprintf( "%04d", 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:

```
string code;

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 "
    "number \"" + 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 number "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 our third 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.

6.4. Quest Utils

In our mudlib is an object called `/obj/misc/quest_info_utils`, which unsurprisingly provides various utility functions to make it easier to deal with quest-related information. You can inherit it along with the rest of your inherits, and to save you from having to hardcode its path, you can include `library.h` which will then allow you to refer to it as `QUEST_UTILS`:

§§§§ Well, *we* have it stored, but we have no way of ensuring that the *player* has it stored. If they don't keep logs, and they fail to write down their number, then maybe they will forget it and be unable to progress in the next quest. We should therefore make sure there's some way they can find out their number again if they really have to. This doesn't have to be an *easy* way — maybe they should have to re-sort the library again to do it — but there should be *some* way. This is left as an exercise for the reader (i.e. you).

```
#include <library.h>

inherit QUEST_UTILS;
```

However, there is a limitation in that it can only be used in an object if that object is involved in one *and only one* quest. Luckily, that's almost all of our quest 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 managing player state information for quests is reduced to simply setting key and value pairs. We use the method `set_quest_param()` to do this:

```
string code;

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.

6.5. Our Second Quest

Let's move on from this quest 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:

- Randomly generate a location for the hole

- If the player has a certain `adventuring.perception` bonus, show the hole when they look in the right place.
- When a player looks at the hole, give them the clues as to how they need to break through it.
- The hole is reached by the player climbing the appropriate bookshelf.
- The player uses the hints given as to tool needed to break through the hole.
- 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 the bookshelf that belongs to a particular category. So let's do this — when the player looks at the categories or the bookshelves, we want to do a perception skillcheck[*****] 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 hole, tool;

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

  switch ( TASKER->perform_task( player, "adventuring.perception", 120,
           TM_FREE ) ) {
    case AWARD:
      player->tm_message( "You feel a little more perceptive.\n" );
    case SUCCEED:
      // Choose a random bookshelf category from all the available categories,
      // and choose a random tool from a list of appropriate tools.
      hole = element_of( query_categories() );
      tool = element_of( ({ "crowbar", "hammer" }) );
      details = ({ hole, tool });
      LIBRARY->set_player_quest_info( player->query_name(), HOLE_QUEST,
        details );
  }

  return details;
}
```

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

***** We discussed using the taskmaster for skillchecks in *LPC For Dummies 1*, where we also instructed you not to concern yourself for now with the choice of `TM_FIXED` as the likelihood of a TM being awarded. Here, we are using `TM_FREE` instead, so now seems like a reasonable time to point you at `help perform_task`, which explains when each of these (and the others, like `TM_COMMAND` and `TM_NONE`, should be used).

```
string books_on_shelves() {
  string *details = check_hole_location( this_player() );
  string ret = "The bookshelves are arranged according to categories.  "
    "Rather tattered-looking labels indicate which category belongs to "
    "which shelf.";

  if ( !sizeof( _sorted ) ) {
    ret += "No books have been sorted onto the shelves.  ";
  } else {
    foreach ( string title, string category in _sorted ) {
      ret += "The book \"" + title_case( title )
        + "\" has been put on the shelf for the category \""
        + title_case( 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", 0,
  "Attempt to break through the roof above a specific 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;
}
```

We have something new here — look at that `add_succeeded_mess()` again:

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

This is a new way of invoking it that we haven't seen before, and also a new `$` code: `$I`. Like `$N` and `$V`, this is replaced with something appropriate when the message is printed to the player (and any onlookers) — in this case, what it's replaced with is the short description(s) of anything included in the second argument, which here is `valid_tools[0]`.

The only thing left to do is to add the command that lets the player climb up through the hole; this is left as an exercise for the reader.

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

6.6. 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!

7. Our Last Quest

7.1. 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 will reveal the answer to the puzzle.

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

7.2. 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 should be logical and 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 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 the player 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 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.

7.3. 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 coordinating 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 the MUD, 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, which 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. Hence none of our other objects has to store this data, because they can 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's no intrinsic reason why things like `perform_task()` couldn't be provided as functions in `/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 you want a set of objects to have access to some common set of functionality, and you use a handler when you want a set of objects to have access to some common set of data. For our purposes, we need a handler.

7.4. The Portrait Handler

We'll create a new subdirectory called `handlers` to put our handler in, and we'll add a define for this subdirectory in our `path.h` too:

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

Now for the handler itself. It's not going to be complicated, and it's not going to inherit from anything, because it's just an object for storing some data and functionality. We start with a `create` method — much like the first look we had at the code for the library:

```
void create() {
}
```

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() {
  _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:

††††† In reality, we wouldn't hardcode the order names here. We'd get them by querying `/obj/handlers/guild_things/wizard_orders` — which is, guess, what, a handler!

```
void add_portrait() {
  class portrait new_portrait;
  string wname, wtitle;
  string *eyes = ({ "blue", "green", "black", "red" });
  string *height = ({ "tall", "short", "medium-height" });
  string *weight = ({ "fat", "skinny", "stocky", "plump" });
  string *looks = ({ "handsome", "ugly", "grotesque", "beautiful" });
  string *hair = ({ "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 ),
    height: element_of( height ),
    weight: element_of( weight ),
    looks: element_of( looks ),
    hair: element_of( hair )
  );

  _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 — this 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:

```
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 will 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, wtitle, hist;
  string *eyes = ({ "blue", "green", "black", "red" });
  string *height = ({ "tall", "short", "medium-height" });
  string *weight = ({ "fat", "skinny", "stocky", "plump" });
  string *looks = ({ "handsome", "ugly", "grotesque", "beautiful" });
  string *hair = ({ "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.",
  });

  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 set up the state of the handler. We're going to have 10 portraits in total, with one of them being the "correct" portrait and two of the others being "suspicious"; this means that as long as a player is paying attention, they will be able to find the correct one within three guesses.

We want the correct and suspicious portraits to be randomly distributed among the list of portraits, since if we always had them in the same positions, word would get around among players that all you have to do is e.g. check the first three portraits, which is not in line with our desire for everyone to have to do the work themselves. So we'll set everything up, and then use the `shuffle()` function to randomise our array:

```
void setup_data() {
  _all_portraits = ({ });

  for ( int i = 0; i < 10; i++ ) {
    switch ( i ) {
      case 0:
        add_portrait( 2 ); // the correct one
        break;
      case 1:
      case 2:
        add_portrait( 1 ); // two suspicious ones
        break;
      default:
        add_portrait( 0 ); // all the others
    }
  }

  _all_portraits = shuffle( _all_portraits );
}
```

Every time we call this method, a new state of the portrait gallery will be set up. We also 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 = ({ });

  foreach ( class portrait this_portrait in _all_portraits ) {
    names += ({ this_portrait->name });
  }

  return names;
}
```

7.5. 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, and 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 our handler 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 parentheses, 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>", 0, "Research a specific subject in "
  "the library" );
```

And the code to handle it:

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

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

  topic = lower_case( args[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;
  }

  research_mess = "$N flick$s through some of the books in the library.";

  add_succeeded_mess( ({
    research_mess + "  As far as you can tell, " + history + "\n",
    research_mess + "\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 parentheses 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 parentheses, this is what the code looks like to the compiler:

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

Due to the way the MUD evaluates these operators, it will attempt to call the function on `portrait_handler` (which doesn't exist), rather than on `/w/your_name_here/betterville/handlers/portrait_handler` (which does). With the parentheses, the code looks like this:

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

The parentheses tell the driver “Hey, put all this together before you attempt to call that function”. This ensures the function gets called on an actual object that the driver knows how to find.

7.6. 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 it greatly simplifies the task of implementing the rest of this quest.

Handlers are a common and powerful approach to development on the Discworld MUD, 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.

8. Finishing Touches

8.1. 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.

8.2. 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" );
  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.";
}
```

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

[‡‡‡‡‡] Note that we haven't provided a way for the player to look at individual portraits, which is a shame, given that our portrait handler has a method that returns descriptions for them. This is left as an exercise for the reader (i.e. you).

```
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.
```

We need to do a little bit of processing so that it actually looks good — capitalising each of the names:§§§§§

```
string portrait_long() {
  string *portraits = ({ });

  foreach ( string name in PORTRAIT_HANDLER->query_portraits() ) {
    portraits += ({ cap_words( name ) });
  }

  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.";
}
```

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, and since we've made sure an attentive player will need no more than three guesses to find the correct portrait, let's set our limit 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.

8.3. I'm Looking Through... Uh... Behind You

Looking behind a portrait is a pretty straightforward 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 whether this portrait had the number-pad behind it:

§§§§§ Note that just like when we used `title_case()` for the titles of the books in section 5.6, we have made sure to limit ourselves to names that are easy to capitalise.

****** The observant will note that we haven't given the player any reason to try looking behind the portraits. This is a problem, because in normal gameplay players are generally just expected to look *at* things rather than also *behind* them, and so requiring them to just magically know that they need to look behind the portraits here violates our principle of consistency. There are a few possible ways to resolve this — you could give a clue in the portrait add_item, or perhaps in a room chat. I'm going to leave this as an exercise for the reader (i.e. you), since it's not incredibly important *how* you do this — it's just important to *do* it, since otherwise this becomes a rather unfair puzzle.

```
int do_look( object *indirect_obs, string dir_match, string indir_match,
             mixed *args, string pattern ) {
  string player_mess, room_mess, wizard;
  int panel;

  wizard = lower_case( args[0] );

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

  player_mess = room_mess = "$N move$s aside a picture and look$s behind "
    "it.\n";

  panel = PORTRAIT_HANDLER->query_portrait_panel( wizard );

  if ( panel == 2 ) {
    player_mess += "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( ({ player_mess, room_mess }) );
  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 put in another add_command to handle entering the code, and make the entrance criterion for that command be 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 ) {
    player_mess += "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.

8.4. Ecretsay Odecay

To begin with, the add_command:

```
add_command( "enter", "code <string>", 0, "Enter a code into something." );
```

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 imagining things!\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 ) {
  if ( !player || environment( player ) != this_object() ) {
    return;
  }
  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!" );
}
```

Wait — `call_out()`? That's something new, so I'd better explain it.

The `call_out()` function allows you to delay calling a function until after a certain number of seconds have passed. Its first argument is the name of the function to call (`"transport_player"` in our code), its second argument is the number of seconds to wait before calling it (`0` here), and any subsequent arguments are passed on to the function (i.e. `transport_player()` in this case) when it is called.

It might seem silly to say we want something to happen in 0 seconds' time, but one important point about call_outs is that they will always happen *after* the current thread of execution has finished. When we `return` from `do_enter()`, the MUD will see that we returned `1` (which indicates success), and hence print the message that we set up with `add_succeeded_mess()`. Our `transport_player()` function will only be called *after* all that has happened.

So using a call_out means we can be sure that everyone will see our player entering the secret code *before* they see that player vanish. If we had simply called `transport_player()` before returning `1`, then the player-vanishing messages would have been printed before the MUD realised that the command had succeeded, and hence before it knew it should print the message about entering the secret code.

Another important point about call_outs is that you can't assume that the state of the MUD will be the same when they are executed as it was when they were called. In particular, we can't assume that our player still exists at the point that `transport_player()` is called — and if they don't exist (e.g. if they had a `quit` command queued and it got executed first), then the `player` variable will contain the value `0`. We therefore add a check of `!player` to make sure we don't cause a runtime error by calling functions on `0`.

Similarly, we also check their environment to make sure they hadn't managed to execute a command that involved leaving the room. You might think "oh, this is magic, it should be able to yoinck them from anywhere", but there are rooms we really *don't* want them to be yoincked from (such as `/room/departures`) so it's simplest just to make sure they're still here in this room that we have complete control over.

8.5. 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!

```
int do_look( object *indirect_obs, string dir_match, string indir_match,
             mixed *args, string pattern ) {
  string player_mess, room_mess, wizard;
  int num_guesses, panel;

  wizard = lower_case( args[0] );

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

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

  num_guesses += 1;

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

  if ( num_guesses > 3 ) {
    add_succeeded_mess( "$N try$s to move one of the portraits aside, but "
      "it refuses to move!  How odd.\n" );
    return 1;
  }

  player_mess = room_mess = "$N move$s aside a picture and look$s behind it.";
  room_mess += "\n";

  switch ( num_guesses ) {
    case 1:
      player_mess += "  The painting moves easily, but there is an ominous "
        "glow that appears and then fades away...\n";
      break;
    case 2:
      player_mess += "  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:
      player_mess += "  It is almost impossible to move the portrait.  It's "
        "like it has suddenly become magically glued to the wall!\n";
  }

  panel = PORTRAIT_HANDLER->query_portrait_panel( wizard );

  if ( panel == 2 ) {
    player_mess += "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 );
  }

  add_succeeded_mess( ({ player_mess, room_mess }) );
  return 1;
}
```

Now each player gets a maximum of three attempts per reset to find the secret panel. 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 if you're interested then I encourage you to spend time looking at how you 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.

8.6. 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 how to write quests. What we've learned is how to write these specific quests, and how to 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 snowflakes. 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've discussed here have shown us examples of all of these tools being used in concert to achieve a fairly complex end.

8.7. 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!

9. The Young Romantics

9.1. 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 young women 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.

9.2. The Young Romantic Template

Because our young women 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 young-romantic NPCs, 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. We’ll create an NPC inherit that will be used for all our Betterville NPCs, just in case we want to add any common functionality to the various inhabitants of the area.

```
inherit "/obj/monster";

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

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

Setting up the NPC involves us making use of some randomised elements — we want to set the long description based on the location the young woman is from, and the name and short based on her 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 );
```

Let's also declare a string that will hold our description as we build it up bit by bit:

```
string str;
```

Now we can get going! We first want to set the name and adjectives 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. We'll start by using the `explode` efun to turn our string into an array, with each word in the string becoming an element in the array (i.e. splitting our string at the spaces).

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

We can get the last element of this array by using the array indexing system with the < operator, as described in the chapter on arrays in *LPC For Dummies 1*:

```
set_name( arr[<1] );
```

And all the other parts by adding in the `..` operator, again as previously described:

```
foreach ( string adj in arr[0..<2] ) {
  add_adjective( adj );
}
```

Having done this, we can set up the rest of our NPC:

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

switch ( place ) {
  case "Lancre Town":
    str = "This is a young woman 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 young woman from Genua.  Genua City is full of "
      "opportunities for anyone 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;
}

// We'll add some more info to str in here.

set_long( str );
```

Next, we can make use of the same system that we used for the portraits to describe each of the young women. 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're 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 $height$ and $weight$, with $length$ $colour$ 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 option, we could make use of it in the description string like so:

```
case "Sto Lat":
  str += "This young woman comes from Sto Lat.  In the city of Sto Lat, "
    "$weight$ women like her 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, so you should choose which is most appropriate for your actual needs.

Finally, we need to set up our young women with some chats and nationalities. We set up the nationalities in our switch statements, and include the load_chat() and load_a_chat() calls as usual. Here's one example of a nationality:

```
case "Lancre Town":
  str += "This young woman comes 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" );
  break;
```

And here are our chats:

```
load_chat( 120, ({
  1, "' I could be 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, "' I can't believe those other fools think they might be 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.

9.3. 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 practice 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, `ob`, 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. Note that we must remember to also call the `event_death()` that lives higher up the inherit tree, using the `::` operator.

So, let's make our NPCs a little more responsive to the world around them, They will express their disgust if they see someone killed, and their surprise if they were part of the reason the person 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 young romantics 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?
```

This is a fairly flexible system, but if all we want to do is have an NPC do something special when it dies, and we don't need the parameters from `event_death()`, there's a simpler way to do it — we can just provide a `second_life()` function. This gets executed as part of the MUD's normal 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!
```

†††††† The easiest way to do this is by calling `do_command()`, e.g.:

```
> cal do_command("kill living things except players")living things except players
```

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 young romantics 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 (e.g. via `help event_death`) 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 print a special message when a corpse appears. 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() ) {
    tell_room( this_object(), "Several small insects fly in, all dressed in "
      "black.  They execute a solemn dance around " + ob->the_short()
      + ", and then fly away again.\n" );
  }

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

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

‡‡‡‡‡‡ Why do we do this with `event_enter()` instead of `event_death()`? It's because `event_death()` doesn't get given the corpse object, and we need the corpse for our `the_short()` call. Note also that as things stand, this message will be printed even if someone just picks up a corpse and then drops it again. We can guard against that by only printing the message if `from` is `0`, indicating a freshly-cloned corpse. If a player or NPC drops a corpse in the room, `from` will be non-zero — in fact, it will be a reference to that player or NPC. So just change the check to `if ( !from && ob->query_corpse() )` and all will be well.

```
The upwardly mobile socialite dies.
Several small insects fly in, all dressed in black.  They execute a solemn
dance around the corpse of an upwardly mobile socialite, and then fly away
again.
```

9.4. 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 these corpse messages across the entire village, we can put the above `event_enter()` function into the `betterville_room.c` inheritable (remember to remove the `::event_enter()` call first, since this inheritable doesn't inherit anything so there *is* no `::event_enter()` to call), and then update it:

```
> upd betterville_room.c
Updated /w/your_name_here/betterville/inherits/betterville_room.
```

Great, now let's update our inside and outside room inherits:

```
> upd inside_room.c outside_room.c
/w/your_name_here/betterville/inherits/inside_room.c line 36: Warning:
event_enter() inherited from both
/w/your_name_here/betterville/inherits/betterville_room.c and
/std/room/basic_room.c; using the definition in
/w/your_name_here/betterville/inherits/betterville_room.c. before the end of
file
Updated /w/your_name_here/betterville/inherits/inside_room.
/w/your_name_here/betterville/inherits/outside_room.c line 37: Warning:
event_enter() inherited from both
/w/your_name_here/betterville/inherits/betterville_room.c and
/std/room/basic_room.c (via /std/room/outside.c); using the definition in
/w/your_name_here/betterville/inherits/betterville_room.c. before the end of
file
Updated /w/your_name_here/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.c`, we've put an `event_enter()` function. However, `betterville_inside.c` also inherits `/std/room/basic_room.c`, and `betterville_outside.c` also inherits `/std/room/outside.c`, and both of those already have 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.c` and `outside_room.c`, and making that function handle the execution of the inherited functions. In `inside_room.c`:

```
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.c`:

```
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 defined in all of their inherits, and take charge of it themselves. They tell the MUD to call the inherited `event_enter()` first in `basic_room.c` or `outside.c`, and then in `betterville_room.c`. This is the usual approach to resolving this kind of error. In certain situations more complex functionality may be required, but it all uses the same basic idea — tell the MUD exactly which inherited functions to call when.

9.5. 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.

10. Lady Tempesre

10.1. 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 individuals on the Beast's "to disembowel" list.

10.2. The Lady

Let's begin with Lady Tempesre Stormshadow — she's the most straightforward of the two. She's designed to be both a service NPC and a flavour NPC, and so 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 her code is fairly straightforward.

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 have particularly complicated code, 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 respond 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 the “evil witch” role that we decided for her when discussing 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.

10.3. 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 disincentivise the activity by adding in a punishment when players kill NPCs we'd rather they didn't, or removing any reward associated (e.g. 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. Moreoever, 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.

10.4. 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() {
  _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 equally as often they’re left public, since 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:

```
string query_player_name( mixed player ) {
  if ( objectp( player ) ) {
    return player->query_name();
  } else 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 an expression of confusion. We can then build that into each of our methods:

```
string *_to_punish;

void create() {
  _to_punish = ({ });
}

string query_player_name( mixed player ) {
  if ( objectp( player ) ) {
    return player->query_name();
  } else if ( stringp( player ) ) {
    return player;
  }

  return "Er, what?";
}

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

  if ( member_array( name, _to_punish ) == -1 ) {
    _to_punish += ({ name });
  }
}

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

  if ( member_array( name, _to_punish ) == -1 ) {
    return 0;
  }

  return 1;
}

void remove_to_punish( mixed player ) {
  string name = query_player_name( player );
  _to_punish -= ({ name });
}
```

Now all we have to do is bind our Lady's 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. We'll create the Beast in the next chapter, and then tying him in to the disembowel handler will be left as an exercise for the reader.

10.5. Data Persistence

The next 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, but in this chapter we're going to go for the simplest of them. 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 create from a variable and then turn back into a variable again later. When an object is saved in its entirety, it is saved as a `.o` file that 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 states 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,])
```

The first line of the file tells the MUD that the save file was generated by the bank handler. The rest of the file 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.

10.6. 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 so on, it depends on the directory in which 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 permission to 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" ) :) );
}
```

The `unguarded()` call says to the MUD: "This is as 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's 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() {
  _to_punish = ({ });
  load_me();
}

string query_player_name( mixed player ) {
  if ( objectp( player ) ) {
    return player->query_name();
  } else if ( stringp( player ) ) {
    return player;
  }

  return "Er, what?";
}

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

  if ( member_array( name, _to_punish ) == -1 ) {
    _to_punish += ({ name });
    save_me();
  }
}

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

  if ( member_array( name, _to_punish ) == -1 ) {
    return 0;
  }

  return 1;
}

void remove_to_punish( mixed player ) {
  string name = query_player_name( player );
  _to_punish -= ({ name });
  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. To do it, 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 a gzipped file. For big files, this will save huge amounts of disk space and file I/O at the cost of a little CPU. It's a bargain whatever way you look at it!

10.7. The Nuclear Option

OK, so we've made sure our data will survive a reboot, which is good. But what about situations where our data *shouldn't* survive — for example, if the player refreshes? Refreshing is supposed to give the player a completely clean start, and what kind of a clean start will it be if it includes being disembowelled due to crimes in a previous life? That's more of a dirty start, really.

Data about players that is stored in *the player object itself* (the player's skills, their guild, any properties that have been set on them, etc) will automatically be deleted when a player refreshes or deletes. However, some player data is stored outside the player; for example, bank balances (as noted in section 10.5, these live in the banking handler) and our disembowelling list. The MUD will not automatically delete this data unless we tell it to.

The solution to this involves the refresh handler, which lives at `/obj/handlers/refresh.c` and, despite the name, handles both refreshing and deletion of players. When you get to the point of putting your own live code in game, any code that stores data about a player in a place that is not *on* that player should have a method to remove data for refreshed/deleted players, along the lines of the following:

```
void refresh_delete_callback( mixed player, int mode ) {
  if ( objectp( player ) ) {
    player = player->query_name();
  }

  // Do your data removal here.
}
```

This function will be called with two arguments: first, the player as either an object (if they're refreshing) or a string (if they are being deleted); and second, an int that describes whether this is a refresh or a deletion.[§§§§§§]

The example above has code to handle both the "passed an object" case and the "passed a string" case, but our disembowel handler is already happy to deal with either an object or a string. So we don't need to bother with that, meaning that we end up with a very simple function:

[§§§§§§] Possible values for this int are `PARTIAL_REFRESH`, `TOTAL_REFRESH`, and `PLAYER_DELETED`, all of which are defined in `/include/refresh.h`. (If you're wondering what a partial refresh is: me too, buddy, me too. It doesn't seem to be a thing any more, so let's pretend we didn't see it.) The reason the player is an object when refreshing and a string when deleting is that someone using the refresh command must be logged in to do so, while someone who's being deleted must have previously issued the `delete` command and then not logged in for the past 10 days, so when this fires they cannot be logged in and hence cannot be an object.

```
void refresh_delete_callback( mixed player, int mode ) {
  remove_to_punish( player );
}
```

The only thing remaining is to inform the refresh handler about this function. We won't do that for our handler, since it's not going into game. If and when you need to do it for real, see `help register_refresh` and `help register_delete`.

10.8. 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, what we can do to restore the state of objects when we need it, and how we can make sure data about players doesn't hang around when we want it to be cleared away. Data persistence is not tricky to do — there's not much more to it than we have talked about here. For very complex situations, or very intensive file I/O, we may need to do something a little more tailored, but for most objects and handlers, you now know as much as you will ever need to know.

11. Beastly Behaviour

11.1. Introduction

Now that we've happily implemented our young romantics 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 shredded away every "
    "last trace of humanity, leaving only a core of pure animal rage.\n" );

  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, "@stare sadly"
  }) );

  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!" );
}
```

We have two new functions here: `implode()` and `allocate()`. The first of these, `implode()`, takes an array as its first parameter and a string as its second parameter, and uses the string as glue to join together the elements of the array into one long string. Here, we're using the empty string (`""`) as our joining string, since all we want to do is smush all the array elements together without anything else in between.

The other new function, `allocate()`, lets us create an array of a given size filled with whatever we desire. Its first parameter is how many elements the array will have, and its second is the starting value to give each element of the array.

The result of our code above is that the Beast can now growl 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 li'l 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 which we will then 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, so let's rewrite `animal_growling()` to use our mangling function too:

```
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..0];
  string end = word[<1..<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 something like "Haaarraarrro".

There's a new technique we're using here — take a closer look at the first line of `contort_word()`:

```
  string start = word[0..0];
```

We're familiar with the use of square brackets to refer to elements of an array, and with the `..` operator to refer to a range of those elements, but `word` isn't an array — it's a string. What's going on?

The thing we're aiming for here is to get the first letter of `word` and store it in `start`. To do this, we take advantage of the fact that strings in LPC are actually stored as arrays, with one individual letter of the string at each position of the array.

You might immediately think that `word[0]` will have the first letter of `word`; it does, in a way, but there is a small nuance that requires us to use `word[0..0]` instead.

Try this exec:

```
> exec string word = "Hello"; return word[0]
Returns: 72
```

So `word[0]` isn't a letter, it's a number! This is all down to how the MUD actually represents those 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 from an string (such as `word[0]`), what we get out is the number representing that letter.

Happily, we can make it all work nicely by using the `..` operator to tell the MUD that we are expecting it to give us a string:

```
> exec string word = "Hello"; return word[0..0]
Returns: "H"
```

Anyway, now we bind all of that into the Beast's `add_respond_to_with()` to provide tailored yet randomised responses:

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

And then the function that handles giving the random output:

```
void lilith_response( object speaker, string message ) {
  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( object speaker, string message ) {
  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.

11.2. 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. First, we'll give him the shove command by using `add_known_command()` in his `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. A higher value 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. And this is what will be executed when it does:

```
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.

11.3. 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. The MUD has an inherit that handles all of that stuff, and its name is `/obj/monster/smart_fighter.c` — if we want our beast to be a bit more clever, we can make use of this.

We want the Beast to inherit both `smart_fighter.c` and our own NPC inherit, `betterville_npc.c`, so he'll have access to any cool Betterville NPC code that we add in the future. If we made him do this directly, we'd have to do a bit of fiddling about to deal with the multiple inheritance issue as we did in section 9.4, since not only do both of those inherits have their own `create()` method, but `smart_fighter.c` and `/obj/monster.c` (which `betterville_npc.c` inherits) both have an `event_fight_in_progress()` method and a `stats()` method, so we'd need to disambiguate them all.

Thankfully, it doesn't have to be that complicated in this case. There is an alternative inherit, `/obj/monster/smart_fighter.c`, which deals with disambiguating between `smart_fighter.c` and `monster.c` — and so we can just make `betterville_npc.c` inherit that instead of inheriting `/obj/monster.c` directly, and then the problem goes away:

```
inherit "/obj/monster/smart_fighter_monster";

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

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

"But wait!" you say. "What about our young romantics? They inherit this too, but they are neither very smart nor very fighty, so they shouldn't be smart fighters!" Rest assured — simply inheriting `smart_fighter.c` does not, in itself, add any smart fighting behavior. For that, we need to use the `set_smart_fighter()` function in an NPC's `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 the necessary commands as well as sensible amounts of the necessary skills. 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()` function. 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 spellcasting. If we want 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 spellcasters, 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.

11.4. 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 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. The decimal numbers we use in everyday life work according to a base 10 system — we naturally work in multiples of 10. Binary is a base 2 system — it's constructed of multiples of 2. Both systems work in terms of powers of their base; for example, the decimal number 123 is constructed as follows:

10^2 (=100)	**10^1 (=10)**	**10^0 (=1)**
1	2	3

Working from *right to left*, each column is 10 times the previous column. This number is the familiar 123 gained by the following simple arithmetic:

```
( 100 * 1 ) + ( 10 * 2 ) + ( 1 * 3 ) = 100 + 20 + 3 = 123
```

Binary works exactly the same way except each column is 2 times the previous column:

2^7 (=128)	**2^6 (=64)**	**2^5 (=32)**	**2^4 (=16)**	**2^3 (=8)**	**2^2 (=4)**	**2^1 (=2)**	**2^0 (=1)**
1	0	0	1	0	1	0	1

The number 1001 0101 is represented by the following arithmetic:

```
( 1 * 128 ) + ( 0 * 64 ) + ( 0 * 32 ) + ( 1 * 16 ) + ( 0 * 8 ) + ( 4 * 1 )
+ ( 2 * 0 ) + ( 1 * 1 )
```

The number, then, is 149 in decimal. If you're not already familiar with using binary representations of numbers, it's worth taking some time to practice converting back and forth between binary and decimal until you're sure you understand how it all works.

Bitwise operators allow you to do the familiar AND and OR operators on each bit in a binary number. Like many other languages, LPC uses a single bar (`|`) for a bitwise OR and a single ampersand (`&`) for a bitwise AND. The result of a bitwise AND is the number we end up with if we do an AND operation on each bit of the number — only if *both* bits are 1 will it make it into the final number. As an example, for `22 & 11` we have:

	2^4 (=16)	2^3 (=8)	2^2 (=4)	2^1 (=2)	2^0 (=1)
22 in binary	1	0	1	1	0
11 in binary	0	1	0	1	1
Result of AND	0	0	0	1	0

So the result of a bitwise AND on the decimal numbers 22 and 11 comes out to 10 in binary, which is 2 in decimal. Let's check that with an exec:

```
> exec return 22 & 11
Returns: 2
```

Hurrah! 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:

	2^4 (=16)	2^3 (=8)	2^2 (=4)	2^1 (=2)	2^0 (=1)
22 in binary	1	0	1	1	0
11 in binary	0	1	0	1	1
Result of OR	1	1	1	1	1

So the result of a bitwise OR on the decimal numbers 22 and 11 comes out to 11111 in binary, which is 31 in decimal:

```
> exec return 22 | 11
Returns: 31
```

It is this process at work inside the smart fighter code — each of the defines is a 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 is a new notation for numbers, and it simply tells LPC that we are defining a hexadecimal (base 16) number rather than a decimal number. The reason for defining these numbers in hexadecimal rather than decimal is that the numbers defined in `smart_fighter.h` are intended for use in bitwise operations, and hence need to be thought about in terms of the binary system.

Using actual binary numbers in `smart_fighter.h` would be very longwinded*******, and bitwise operations on decimal numbers are hard for humans to think about. However, the hexadecimal and binary number system share a nice relationship — each digit of a hex number corresponds to four digits of binary††††††† — and so hex numbers are used to define these values, for optimum human-readability.

Anyway! The values in `smart_fighter.h` are chosen so that every combination of possible values has a unique result when bitwise OR is used to combine the... well, the combination. We can later use the bitwise AND operator to see whether or not a specific value is represented. For example, when we use the bitwise OR operator on `USE_SHARP` (hex 0x0008, decimal 8) and `USE_PIERCE` (hex 0x004, decimal 4) — i.e. when we put

```
( USE_SHARP | USE_PIERCE )
```

in our code, we get:

	2^3 (=8)	2^2 (=4)	2^1 (=2)	2^0 (=1)
8 in binary	1	0	0	0
4 in binary	0	1	0	0
Result of OR	1	1	0	0

i.e. our fighter setting is 1100. Later on, if we want to know whether or not someone has set `USE_SHARP`, we would do the following:

	2^3 (=8)	2^2 (=4)	2^1 (=2)	2^0 (=1)
Fighter setting	1	1	0	0
USE_SHARP	1	0	0	0
Result of AND	1	0	0	0

The result is 1000 binary, which is true, so we know that we should treat this NPC as if it were a sharp user. Likewise, if we wanted to check if it used blunt, we would look at the bitwise AND of our fighter setting and `USE_BLUNT` (which is defined as 0x0010):

	2^4 (=16)	2^3 (=8)	2^2 (=4)	2^1 (=2)	2^0 (=1)

******* It wouldn't be too bad for `USE_SHARP` and `USE_PIERCE`, which are 1000 and 100 in binary, respectively, but `JOIN_ANY_FIGHT` is 0x10000, which is 1 0000 0000 0000 0000 in binary.

††††††† Observe: 0x0001 is decimal 1 (binary 1, which fits into four binary digits: 0001), 0x0002 is decimal 2 (binary 10 aka 0010), ... 0x0009 is decimal 9 (binary 1001), 0x000a is decimal 10 (binary 1010), ... and 0x000f is decimal 15 (binary 1111). If we add 1 to that, we get 0x0010, which is decimal 16 and binary 1 0000, aka binary 0001 0000, i.e. we've had to go into a second hexadecimal digit and a second batch of four binary digits.

Fighter setting	0	1	1	0	0
USE_BLUNT	1	0	0	0	0
Result of AND	0	0	0	0	0

Voila, the result of the AND (i.e. zero) indicates that we should not use blunt for this NPC.

All of this is slightly arcane, and you don't *really* need to know how it all works — it's enough to know that it *does* work. However, you'll find this kind of system is used in several places throughout the MUD, and a sound understanding of Why It Is So will serve you well.

11.5. 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 — `set_smart_fighter()` takes up to three arguments,[‡‡‡‡‡‡‡] but since you can combine things with bitwise OR, you have many more than three possible options. Some of the options you have available are:

Flag	Arg #	Description
`USE_COVERT`	1	Use 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	Join in combat if any combat is ongoing in the room

‡‡‡‡‡‡‡ Technically it takes up to four, but the fourth argument is not completely implemented, so unless that situation has changed between when I'm writing this and when you read it, it's not worth bothering with.

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.

11.6. 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 in 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 the 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 or both, then we can do:

```
string *arr3 = arr1 | arr2;
```

This would give us the array `({ "a", "b", "c", "d" })`.

Alas, we currently have no way of doing a bitwise operation that gets elements that are in one array or the other but not both — but we never know what lovely things the next driver update will bring us.[§§§§§§§]

11.7. 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.

§§§§§§§ There is an exclusive or (XOR) bitwise operator (^), but at the time of writing it doesn't work on arrays.

The Beast is now complete, as much 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!

12. Shopping Around

12.1. Introduction

There's a feature we haven't yet addressed in Betterville — our shop catering to the whims of the young romantics. That's a major deficiency as things stand at the moment, because they 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're 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.

12.2. Item Shop Inherit

First of all, we need an inherit for our shop. We want the shop to inherit both our core Betterville inherit and the mudlib's item shop inherit, and as we know by now, this means we will have to disambiguate any multiply-inherited functions — `create()`, for example.

Now technically we could put this disambiguation in our shop itself, but that's bad practice — not only is it unusual to have a `create()` function in a file that isn't an inherit, but if we decided later on that we wanted to have a second shop in the village, we'd need to either copy-paste all the disambiguation into it (which is very naughty) or make a "Betterville item shop" inherit anyway. So we may as well make the inherit now. We can make a start like so:

```
#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();
  }
}
```

When we `update` the above, we will get warnings about three other functions that are inherited from more than one place: `dest_me()`, `event_theft()`, and `init()`. The `dest_me()` and `init()` methods are straightforward to deal with:

```
void dest_me() {
  inside_room::dest_me();
  item_shop::dest_me();
}

void init() {
  inside_room::init();
  item_shop::init();
}
```

However, `event_theft()` 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.

12.3. 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 set up 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 "
    "women.  The garments are each fit for a princess, with price tags "
    "to match.\n" );
  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!

12.4. Item Development

We spoke about clothes and virtual files in the *LPC for Dummies 1*, 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/` subdirectory of Betterville. Unfortunately, this doesn't work as well as we might hope — the armoury will not pick items out of subdirectories 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 subdirectory 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 instead navigate across two different directories.

We're not going to do that — we're going to be awkward and store things in `/betterville/items/`. This is purely something we're doing to make these tutorials hang together properly — when developing code in the "real world", it should always be accessible from the armoury.

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_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're going to add that dress to the shop in the Usual Manner; that is, by putting the following 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.

12.5. Dressing Up

Let's say I want these dresses to be a little more configurable than normal dresses. As well as their short and 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:

```
inherit "/obj/clothing";

string _motif;

void setup() {
  set_name( "dress" );
  set_short( "beautiful blue dress" );
  add_adjective( ({ "beautiful", "blue" }) );
  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 extra_look( object ob ) {
  if ( _motif ) {
    return "The dress is decorated with a lovely motif of " + _motif + ".\n";
  }
  return "";
}
```

Now when I clone one and call `set_motif( "dancing monkeys" )` on it, I get:

```
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 is decorated with a lovely motif of dancing monkeys.  It is
in excellent condition.
```

Lovely, just what I've always wanted. Alas, it is ephemeral, like the love of a young woman. 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 want the motif to persist over logout. The MUD, unfortunately, does not. This is where we need to hook into a clever system known as *autoload*.

12.6. Autoload

When a player saves, the MUD goes through a process of taking their skills, tell history, health, guild points, and all the other things that make up that unique player and storing that info in a file on disk. As part of that process, it also stores their inventory.

When an item is stored, the MUD needs to store not only the item's filename, but also any elements of the item that can vary depending on its configuration (such as the colour, style, and so on of custom items) as well as any elements that are subject to change as the game goes on (such as enchantment levels, condition, engravings, and so forth). This complicated procedure is known as the *autoload* 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 autoload process.

There are two types of autoload data: static and dynamic. Static data should be used for parameters that vary across instances but are static over time (e.g. our dress's motif, or a custom item's configuration), while dynamic data should be used for parameters that can change over time (e.g. enchantment level).[********]

In this case, because our motif is static over time — once motifed, always motifed — we want to save it as static data. Let's start by specifying the data we want to save:

```
mapping query_static_auto_load() {
  return ([
    "motif" : _motif,
  ]);
}
```

We don't need to call this method at any point; it gets called naturally as part of the saving and reloading processes that occur when we log off and on.

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, but it's not being restored, and the reason for this is that each object is also responsible for restoring its own state. For this, we use the method `init_static_arg()`. As with `query_static_auto_load()`, this method gets called automatically as part of the autoload process, along with all the parameters we will need. The first parameter is the mapping that contains all of the autoload information, and the second contains the player for whom the inventory is being loaded.

```
void init_static_arg( mapping map, object ob ) {
  if ( map["motif"] ) {
    _motif = map["motif"];
  }
}
```

And that's that!

******** For an explanation of why the distinction between static and dynamic data is important, see `help autoload`. You will, unfortunately, also probably come across code written by people (including the author of the present footnote) who didn't understand this distinction, and hence who used the wrong type. It's tricky to fix this sort of mistake after the fact, so make sure you get it right when you write your own code.

12.7. 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, and 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 our player to get a delicious melon:

```
object create_object( string item ) {
  object ob;
  ob = ARMOURY->request_item( "melon" );
  return ob;
}
```

Now when we list the stock, all that is for sale is melon:

```
The following items are for sale:
  A: a juicy melon for 2,66Gl (five left).
```

That's no 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 ) );
```

So let's do that, and then in our `create_object()` we can use a `switch` statement to handle the creating of the appropriate item and the setting of the appropriate motif:

```
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 to see the difference between the two, except by browsing them, and we have not provided any great incentive for monkey (or pirate) fans to do so. It's also a bit of a hassle to have the long string `"beautiful dress with a blah motif"` repeated throughout our switch statement. Luckily, this is all fixable by using a different format of `add_object()`, 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 parameter 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 items within a shop — you're not limited to stocking *only* things customised with `create_object()` or only things *not* customised with `create_object()`. When the shop code gets to the point of needing to create an object in its stock, it does the following:

- Call `create_object( name_of_thing )`
- If that returns `0`, call `clone_object( name_of_thing)`
- If that returns `0`, call `ARMOURY->request_item( name_of_ thing)`
- If that returns `0`, give up

So if we want to stock melons as well as dresses, we can do that with the normal `add_object()` call:

```
add_object( "melon", 8 );
```

We will also need to amend our `create_object()` function so it only returns a dress if a dress is definitely what was requested:

```
object create_object( string item ) {
  object ob;

  // We only handle dresses.
  if ( strsrch( item, " dress" ) == -1 ) {
    return 0;
  }

  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;
}
```

There's a new function in here — `strsrch()`, which searches through a string to see if it contains a specific substring. If it does, then the return value is the position in the string at which the substring can be found; if it doesn't, then the return value is `-1`. So our `create_object()` is checking whether our item contains the string `" dress"`; if it doesn't, i.e. a dress is *not* what has been requested, then our `create_object()` returns `0`.

And then when someone comes to buy a melon, the MUD does this:

- Call `create_object( "melon" )`, which returns `0`, because the string `"melon"` does not contain the substring `" dress"`

- Call `clone_object( "melon" )`, which also returns `0`, because there is no item with a filename consisting entirely of `"melon"`
- Call `ARMOURY->create_object( "melon" )`, which finally returns us a beautiful melon, so the process stops here

If we want to stock dresses without motifs, we do:

```
add_object( ITEMS + "beautiful_dress", 3 + random( 3 ),
  "beautiful but unadorned dress" );
```

And then the MUD proceeds like so:

- Call `create_object( ITEMS + "beautiful_dress" )`, which returns `0`[††††††††]
- Call `clone_object( ITEMS + "beautiful_dress" )`, which returns our dress, so the process stops here

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.

12.8. 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, which is bad for all sorts of reasons).

The autoload system is an extremely flexible way of doing this. It also puts the responsibility for proper storing and restoring of state information in the hands of individual creators, where it belongs.

†††††††† This is why we made `strsrch()` check for `" dress"` rather than just `"dress"`. The check is still not foolproof; for example, if we had a "smart dressing gown" in stock then we would see erroneous matches. However, we *know* we don't have one, because we have complete control over this shop's stock, so all is well. If we did decide to stock a dressing gown, we'd have to make our check a bit more complicated; see `help regexp` for one way of doing it.

13. Spelling It Out

13.1. 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. You might never end up coding a new spell of ritual at any point in your creating career, but it's well worth knowing how they are put together and how they function — even if only for bugfixing purposes. 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 we like during development, and so we are going to keep ours in a Betterville subdirectory called `/magic/`, like so:

```
#define MAGIC BETTERVILLE + "magic/"
```

So, with no further ado, let's a-do it!

13.2. The Anatomy of a Spell

At its simplest level, a spell is simply a set of configuration details along with code that happens if the spell succeeds, and code that happens if the spell fails. Spells can get a lot more complicated than this, but they don't have to!

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( ({ }) );
}
```

That's quite a lot of information to provide, but the process of providing it is fairly straightforward. 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 mindspace 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. Here, "direct" doesn't mean the same thing as it does in `add_command()` — instead, 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, but the skill we provide here is used 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:[‡‡‡‡‡‡‡‡]

[‡‡‡‡‡‡‡‡] Note that the process looks slightly different for witch spells, since they have their own inherit that deals with crystals and so on, and so the syntax for providing information about the stages is different too.

```
set_ritual(
  ({
    ({
      ({
        "n/a",
        "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,
      ({ })
    }),
    ({
      ({
        "n/a",
        "You wiggle your fingers suggestively in the air in front of "
          "$ob_name$.\n",
        "$tp_name$ wiggles $tp_poss$ fingers suggestively in the air.\n",
        "$tp_name$ wiggles $tp_poss$ fingers suggestively in the air in "
          "front of you.\n"
      }),
      "magic.methods.physical.evoking",
      140,
      ({ })
    }),
    ({
      ({
        "n/a",
        "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,
      ({ })
    }),
    ({
      ({
        "n/a",
        "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,
      ({ })
    }),
    ({
      ({
        "n/a",
        "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:

```
      ({
        "n/a",
        "You wiggle your fingers suggestively in the air in front of "
          "$ob_name$.\n",
        "$tp_name$ wiggles $tp_poss$ fingers suggestively in the air.\n",
        "$tp_name$ wiggles $tp_poss$ fingers suggestively in the air in "
          "front of you.\n"
      }),
      "magic.methods.physical.evoking",
      140,
      ({ })
```

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
}),
```

This spell cannot be cast on its caster, so the first of these messages will never be used. We put `"n/a"` here (for “not applicable”). You could also use the empty string, or you could even copy one of the other messages — it doesn’t matter, since no player should ever see this. You’ll see the technique of copying one of the other messages in a lot of our older spells, but these days we usually use `"n/a"` or the empty string for any messages that will not be printed, to make it clear what’s going on.

The other three messages are all different, because we’re going to make it so that this is the only stage that informs the target that they *are* in fact the target, and only the caster themself and the target will get this information. We’re doing this for illustrative purposes, but it’s a technique you can make use of for narrative or game-balance purposes too.

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 do whatever is necessary for the spell to function 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 spellcasting was.

For our succeeded effect, let’s give the 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:

```
#include <tasks.h>

[...]

void spell_succeeded( object caster, object *targets, int bonus ) {
  int ret;
  // This spell cannot be cast against more than one target.
  object target = targets[0];

  // spell_succeeded() is called with a call_out from the base magic spell
  // code, so we need to make sure our target is still here.
  if ( !target || environment( target ) != environment( caster ) ) {
    tell_object( caster, "Your target seems to have disappeared!\n" );
    return;
  }

  ret = TASKER->perform_task( target, "magic.methods.spiritual.abjuring",
    bonus, TM_FREE, 0 );

  switch ( ret ) {
    case AWARD:
      target->tm_message( "You feel more capable of resisting hostile "
        "magic!\n" );
    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...\n" );
      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, since 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 in front of Lady Tempesre
Stormshadow.
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!

13.3. More Complex Spells

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(
    ({
      ({
        ({
          "n/a",
          "You start to click your fingers.\n",
          "$tp_name$ starts to click $tp_poss$ fingers.\n",
          "n/a"
        }),
        "magic.methods.physical.dancing",
        130,
        ({ "#check_room" })
      }),
      ({
        ({
          "n/a",
          "You do a little dance with a beautiful blue dress.\n",
          "$tp_name$ does a little dance with a beautiful blue dress\n",
          "n/a"
        }),
        "magic.methods.physical.dancing",
        200,
        ({ "a beautiful blue dress" })
      }),
      ({
        ({
          "n/a",
          "You throw the dress up into the air.\n",
          "$tp_name$ throws the dress up into the air.\n",
          "n/a"
        }),
        "magic.methods.spiritual.summoning",
        180,
       ({ })
     })
    }) );
}
```

Note here for stages one and two we're doing something slightly different — the fourth piece of data, which was an empty array for our earlier spell, now contains some information.

In stage one, this information is `({ #check_room })`, which tells the code that we want to call the function `check_room()` to see whether or not the spell should progress beyond this stage. This function should return `1` on success and `0` on failure:

```
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;
}
```

Wait a minute, what's that `query_lady_tempesre_clothes_shop()` function? We haven't defined that yet, so we should probably do so. Add the following to the code of the shop:

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

Now, if the caster is inside the shop when `check_room()` is called, the function above will return `1` when called, and the spellcasting can continue. If the caster is anywhere else, the room they're in will not have a function called `query_lady_tempesre_clothes_shop()`, and so the call will return `0` and the spellcasting will fail.

In stage two, the extra information is `({ "a beautiful blue dress" })`, and this 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.

We also have our `spell_succeeded()` and `spell_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( environment( caster ), "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;
}
```

We have another new function here: `make_corpse()`. 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 dress NPC comes with an expiry date — they get a `call_out` to `do_death()` 30 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. Together, these two spells make her capable of dealing with those who see fit to disturb her commerce.

13.4. 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 ) {
  object env = environment( this_object() );

  if ( !env ) {
    return;
  }

  if ( !_spare_dress && env->query_lady_tempesre_clothes_shop() ) {
    tell_room( env, 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 spellcasting and at least give people a chance to defend against her wrath. However, this doesn't stop her from casting two spells at once — for that, we'll add a call to `query_casting_spell()` in each function:

```
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:

```
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 for other young women to follow — don't mess with Our Lady.

13.5. Conclusion

Although you might never end up writing your own spell, you should still know how they are put together so that you can make your NPCs use them effectively, help fix any bugs in them, and add more functionality to them if this is ever deemed necessary.

Writing NPCs that use spells intelligently 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.

14. Cause and Effect

14.1. Introduction

In the last chapter we looked at spells — but they were rather simple spells that just had one immediate result. Often spells act as a kind of delivery engine for an effect, rather than being something that instantly delivers a payload. In this chapter we're 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 easy enough to implement, but we're going to make the searching a little more interesting than we have done in the past.

14.2. What's An Effect?

The easiest way to think of an effect is as a temporary set of conditions that impact 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.

Some effects come in 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 overrides functions like `short()` and `long()` to make a player appear to have been turned into a frog, or a shadow that overrides `query_skill_level()` to simulate an object that gives bonuses to certain skills. But we aren't going to discuss shadows here, since most of the things that you'll want to do with an effect can easily be done without adding a shadow too.[§§§§§§§§]

What we're going to do in this chapter is build an effect that acts as a "damage over time" system on a player. As a result of searching for the hidden path, they'll get scratched and damaged, and they will then bleed for a while. It's a simple effect, but that that will illustrate a good deal about how effects are put together.

§§§§§§§§ You may find pieces of advice or documentation telling you that shadows are deprecated and that no new shadows should be coded. This was true once, but shadows have now been rehabilitated and so it's once more OK to code them. You should still avoid using them unless you really need them, since they can make bug-hunting more difficult, but if you need one, it's OK to write one.

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>

// This is called when the effect is first added to an object.
mixed beginning( object player, mixed args, int id ) { }

// This is called when the effect is added to an object that already has the
// same effect active.
mixed merge_effect( object player, mixed old_args, mixed new_args, int id ) { }

// This is called when the effect is removed from an object.
void end( object player, mixed args, int id ) { }

// This is called when a player logs off and on again.
void restart( object player, mixed args ) { }

// This provides an "identifier" code for the effect.
string query_classification() { }
```

Let's start by filling in a placeholder for each of these functions and discussing what their parameters do.

```
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 );
  return args;
}
```

The first parameter to `beginning()` is the object to 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.

The second parameter contains any arguments that were provided when we added this effect to this object; these are generally used for setup and configuration purposes. In our example above, the arguments configure the amount of damage we apply.

The third parameter is the ID of the effect; you will also see this referred to as the "enum" (short for "effect number"). We'll explain this below when we discuss the `effect` command.

When we return a value from `beginning()`, the value we return is stored on the player object as the "args" of this effect (note: this does not *have to* have the same value as the arguments passed as the second parameter to `beginning()` — it doesn't even have to be the same type of variable or data structure*********). We can then access it in other places in

********* If you're now wondering whether we are in fact using the term "args" to mean two different things — (a) the initial configuration data passed into an effect, and (b) the internal data that the effect wishes to store on the player — then congratulations, you are correct. Unfortunately this confusing terminology is present in the vast majority of effects that have ever been coded for the MUD, so we're going to use it here as well so you can get used to it.

our code; we'll come back to this later. For now, all our effect does when it is applied to a player is to do an amount of damage equal to the argument we pass it.[†††††††††]

By default, if two effects are added to the same object, the object gets two copies of an effect running on it. Usually we don't want that, and so we override that behaviour with the use of `merge_effect()`. This function is passed four parameters: the object, the args that are currently stored on that object, the arguments that were provided when the effect was added for the second time, and the ID of this effect on that object. Whatever we return from this function becomes the new value of the args 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;
}
```

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. Three parameters are provided: the object, the args stored on that object, and the ID.

```
void end( object player, mixed args, int id ) {
  tell_object( player, "The stinging from the nettles subsides.\n" );
}
```

Since `restart()` gets called each time the effect gets restarted, let's use it to supply a message to the player:

```
void restart( object player, mixed args ) {
  tell_object( player, "You can still feel a stinging sensation from those "
    "stinging nettles you foolishly put your hands into.\n" );
}
```

For `query_classification()`, we give some meaningful identifier for this effect. It doesn't really matter what it is, but by 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";
}
```

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:

††††††††† There are two things missing from this implementation of player-damaging, namely that whenever you take HP off a player, you need to (a) check whether you should show their combat monitor (i.e. the `Hp: 400 (2000) Gp: 15 (200) Xp: 888` thing) and (b) check whether you have in fact killed them. We'll cover this in section 14.6, after we finish talking about effects.

```
> call add_effect( "/w/your_name_here/betterville/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, we 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. This is the ID aka enum aka effect number I mentioned earlier — the `int` that gets passed around to the functions in our effect. 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 parentheses indicates the args of that effect — in this case, a simple number, but effects with more complicated args will show the string `"COMPLEX"` instead. This effect will remain in place until I die, or until I delete it like so:

```
> call delete_effect( 1 ) me
The stinging from the nettles subsides.
```

I can add this effect to myself as many times as I like, 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. Now let's open this puppy up and see what we can do.

14.3. 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. Our nettle effect currently just does the damage once and that's it. If we want damage to also be done on a periodic basis, we need to use the `submit_ee()` function on our player ("ee" is short for "effect event"). This function takes three parameters: the first is the function to be called when the effect event occurs, the second is the interval (in seconds) between the "ticks" of the event (only relevant if it needs to be called more than once), and the third is how often and how many times the event should be called. This third argument should be one of three values, as defined in `effect.h`:

Effect Interval	Description
`EE_REMOVE`	Call it once, and 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 to happen continuously while the effect was active, that's very easily done:

```
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;
}
```

This will call the function `damage_me()` every 20 seconds, passing it our three familiar parameters: the object, the args stored on the object, and the ID:

```
void damage_me( object player, mixed args, int id ) {
  int damage = random( args );
  // Let's make sure the damage is at least _slightly_ noticeable.
  if ( damage < 5 ) {
    damage = 5;
  }
  player->adjust_hp( -1 * damage );
  tell_object( player, "Oooh, that stings!\n" );
}
```

Every 20 seconds, our effect does 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 call to `submit_ee()` with `EE_REMOVE` as the third parameter. No need to give it 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 code more flexible by passing in an array something like `({ damage_amt, time_to_last })` 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. Our `merge_effect()` is already handling the possibility of a player being affected by a stronger version of the effect (remember, whatever we return from `merge_effect()` becomes the new args of the effect), but we should also make sure it properly figures out any changes needed to the duration. That's easy to do with the `expected_tt()` function:

‡‡‡‡‡‡‡‡‡ See footnote on page 144 regarding what's missing here, as well as section 14.6. Note also that if we didn't have our minimum 5 damage, it would be possible for this to give zero damage (since `random( args )` returns a value between `0` and `args - 1` *inclusive*), and so we would also need to handle that case separately.

```
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 fixed 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 our `betterville_02.c` room 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." );

  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, success;

  success = TASKER->perform_task( this_player(), "adventuring.perception",
    100, TM_FREE );

  switch ( success ) {
    case AWARD:
      this_player()->tm_message( "You feel a little more perceptive.\n" );
    case SUCCEED:
      tell_object( this_player(), "You see a trail leading through "
        "the brush to the southeast.\n" );
      found = 1;
  }

  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 have `/std/basic/effects` somewhere in their inheritance tree. As such, effects can exist on inanimate objects as easily as they can on players, and the mechanisms for dealing with them are identical.

14.4. 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; we pass it the classification of the effect we want to look for, and it gives us back an array of all the matching enums, 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, success;
  int *enums;

  enums = this_player()->effects_matching( "body.nettles" );

  if ( sizeof( enums ) ) {
    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;
  }

  [...]
}
```

Another thing we often want to do is remove effects based on some external factor. An old folk remedy for nettles claims that *Rumex obtusifolius* — otherwise known as dock leaf — is effective for removing the stinging sensation. So we'll add some dock leaf here and let players use it to remove the nettles effect. We'll use `delete_effect()` to handle that. First, the add_item:[§§§§§§§§]

```
add_item( ({ "dock leaf", "dock" }), ({
  "long", "This plant has large oval leaves with gently scalloped edges.  "
    "According to folklore, rubbing the leaves on a nettle sting can help "
    "alleviate the stinging sensation.",
  "rub", ({ "Rub some dock leaf on yourself.", (: dock_leaf_function :),
          "<direct:object'leaf'> [on] [me]" }),
 }) );
```

And then the function that does the magic:

§§§§§§§§ The `"rub"` entry here is in a new format we haven't previously discussed in terms of add_items, but its components should be familiar to you from our discussion of `add_command()`. The first thing in the array is the syntax details, the second is the function to call (see chapter 15 for more on that), and the third is the command pattern.

```
int dock_leaf_function() {
  string player_mess;
  int *enums = this_player()->effects_matching( "body.nettles" );

  if ( sizeof( enums ) ) {
    // We've made sure the player will only ever have one copy of this effect
    // on them, so we don't need to use a loop here.
    other_call_out( this_player(), "delete_effect", 0, enums[0] );

    player_mess = "Throbbing from the stinging nettles, you rub a dock leaf "
      "over your inflamed skin.\n";
  } else {
    player_mess = "You rub a dock leaf over yourself, but nothing seems to "
      "happen as a result.\n";
  }

  add_succeeded_mess( ({ player_mess,
    "$N rub$s a dock leaf over $oself and looks hopeful.\n" }) );
  return 1;
}
```

Note the use of `other_call_out()` — this is like `call_out()` but it calls a function in another object rather than in the current object. We do this here for the same reason that we used `call_out()` in section 8.4 — we know that the nettle effect will print a message to the player when `end()` is called on it, and we want that message to come *after* our message about rubbing.

Now, the thing about folk cures is that they are horribly ineffective. Really, our dock leaf 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 args with `arg_of()`, and we can change their value with `set_arg_of()`. Both of these need us to provide the enum of the effect we wish to query or modify:

```
int dock_leaf_function() {
  string player_mess;
  int *enums = this_player()->effects_matching( "body.nettles" );

  if ( sizeof( enums ) ) {
    int val = this_player()->arg_of( enums[0] );
    val = val - 25;

    if ( val <= 0 ) {
      other_call_out( this_player(), "delete_effect", 0, enums[0] );
    } else {
      this_player()->set_arg_of( enums[0], val );
    }

    player_mess = "Throbbing from the stinging nettles, you rub a dock leaf "
      "over your inflamed skin.\n";
  } else {
    player_mess = "You rub a dock leaf over yourself, but nothing seems to "
      "happen as a result.\n";
  }

  add_succeeded_mess( ({ player_mess,
    "$N rub$s a dock leaf over $oself and looks hopeful.\n" }) );
  return 1;
}
```

For complicated effects, the args may well be more complex than just an integer, but the principle is identical. The `arg_of()` function gives us the args of the effect, and `set_arg_of()` allows us to change them.

14.5. 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 thing is that 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 the 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, but it's entirely possible that such an effect might not have any recurring 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 from 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 need to 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 parameters: the object on which the effect is active, the args of the effect, the ID of the effect, 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 args, int id, int skill_bonus ) {
  if ( skill_bonus > args ) {
    return 1;
  }

  return 0;
}
```

14.6. The Combat Monitor

This information is not specific to effects, but I have to put it somewhere, so I'm putting it here.

As we noted earlier (see footnotes on pages 144 and 146), whenever you take HP off a player you should do two things:

- check whether you should show their combat monitor
- check whether you have in fact killed them

The first of these is very simple:

```
if ( player->query_monitor() ) {
  player->monitor_points();
}
```

The second is a little more complicated. You need to check whether the damage is enough to kill the player, and if so then call `attack_by()` to register that it's this effect that's killing them (and you might also want to give them a different message in this case). You don't need to explicitly call `do_death()` to kill them — reducing their HP to ≤ 0 is enough.

```
void damage_me( object player, mixed args, int id ) {
  int damage = random( args );
  // Let's make sure the damage is at least _slightly_ noticeable.
  if ( damage < 5 ) {
    damage = 5;
  }

  if ( damage >= player->query_hp() ) {
    player->attack_by( this_object() );
    tell_object( player, "Oooh, that stings.  Terminally.\n" );
  } else {
    tell_object( player, "Oooh, that stings!\n" );
  }

  player->adjust_hp( -1 * damage );
  if ( player->query_monitor() ) {
    player->monitor_points();
  }
}
```

Note that you'll need to do this for the initial `adjust_hp()` in `beginning()` as well.

You should also write an amusing death message that will be printed to all online creators who have death informs turned on when the player dies:

```
string query_death_reason() {
  return "grasping the nettle a little too enthusiastically.";
}
```

14.7. 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. 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" effect that incorporates all sorts of deleterious consequences. 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.

15. Function Pointers

15.1. 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 so the next step 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.

15.2. 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 parameters 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:

```
   > exec function f = (: $1 + $2 :); return evaluate( f, 10, 20 )
   Returns: 30
```

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 book add_item from the library:

```
add_item( "book", (: books_to_sort :) );
```

What we've done here is create a function pointer that binds to the locally defined function `books_to_sort()`. 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 recognise 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 overstated — it lets you bundle an awful lot of functionality into a very small space if you know what you're doing.[**********]

[**********] For more information about function pointers, see the creator wiki page `Documentation - Function Pointers`.

15.3. The Four Holy Efuns

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. These 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()`. We've already met the first two, so let's start by getting acquainted with the other two.

Firstly, `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 as 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. Then the MUD steps 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 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 to the `interactive()` efun. Those objects that return `1` get returned from the filter.

The sister function of `filter()` is `map()`, and it works similarly — 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 here is to loop through the array provided by `users()`, call `query_name()` on each element, take the result of that function call and add it to an array to be returned.

Now, here's where we start getting a bit adventurous. The real power of these efuns comes 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() :) )
```

A warning: code like this can rapidly become 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.

When we used `implode()` in section 11.1, we passed it a string as its second parameter, but it will also take a function pointer as its second parameter. When we do this, it takes the first and the second elements 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:

```
({ 10, 20, 30, 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 final step, it gets the results of the evaluation plus the last remaining element:

```
60 + 40
```

The result of this `implode()` is the value 100 — essentially, it sums up all of the elements in the array in a quick, elegant. way.

Conversely, `explode()` doesn't permit the use of a function pointer, but it's the natural companion to `implode()` and hence has earned its place as 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"
```

I want to check to see if these people are online, and if they are display their properly capitalised names. I can do that with a function:

```
string fix_names( string str ) {
  string *names = explode( str, "," );
  string *online = ({ });
  string ret;

  foreach ( string name in names ) {
    if ( find_player( name ) ) {
      online += ({ cap_words( name ) });
    }
  }

  ret = implode( online, ", " );
  return ret;
}
```

Or I can do it in a much more compact way with `implode()`, `map()`, and `explode()`:

```
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 the latter 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 they should not be over-used. 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.

15.4. 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 we write 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";
}
```

Note that this provides a *completely* dynamic long, where the description is recalculated every time someone looks at it. If the state of the library is going to change a lot, then it's fine. If it's only going to change once, or only very occasionally, then we'd need a different strategy, for reasons which I will now explain.

Internally, the argument we pass to `set_long()` 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 means that if `long_d` is a function pointer, then that function pointer is evaluated *every time* the `query_long()` function is called — which, as I'm sure you can imagine, isn't great for the efficiency of your code, 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.

The alternative is to use the *return value* of our function as the parameter to `set_long()`:

```
set_long( library_long() );
```

†††††††††† If you're wondering whether `long_d` is a global variable and hence should have a leading underscore — yes, it is, and yes, it should. This is old code that was written before we had that convention. I have also improved the formatting of the function itself in the version presented here, in order to make it clearer.

This is evaluated only when `set_long()` is called, and whatever comes out of that function is stored in `long_d`. The question you need to ask yourself is: will this room (or object) get changed more often than it gets looked at, or will it get looked at more often than it gets changed?

It is exactly this same principle at work when we set a function pointer to be evaluated in an add_item:

```
add_item( "book", (: books_to_sort :) );
```

That function gets evaluated each time the player looks at the book add_item. Again, it comes at a cost, but we can take that hit provided it's not over-used.

15.5. 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 the third parameter to add_command:

```
foreach ( string verb in ({ "browse", "read" }) ) {
  add_command( verb, "blurb on <string'book'>", (: do_read :),
    "Read the blurb on the back of a book." );
  }
}
```

Note that if we didn't put the `(: do_read :)` here, then if a player used the `browse` verb, the MUD would try to call a nonexistent function called `do_browse()`, which would produce an error. When using a function pointer in this way, we always need to prototype the function (or put it above `init()` in the file) 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, our `do_read()` function doesn't care about anything other than the book title the player entered, 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":

```
foreach ( string verb in ({ "browse", "read" }) ) {
  add_command( verb, "blurb on <string'book'>", (: do_read( $4[0] ) :),
    "Read the blurb on the back of a book." );
  }
}
```

And now, rather than our complicated list of parameters, we just have one string coming in:

```
int do_read( string title ) { }
```

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 function 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. When a chat or response is selected, the MUD 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. You can see this for yourself by looking at `expand_mon_string()` in `/obj/monster.c` (use the find command).

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, they are difficult to debug in certain cases, and they do not benefit from any specialised functionality that comes from more tailored support in the mudlib.

15.6. A Few More Things About Function Pointers

You can pass parameters to a function pointer, but when doing so you can't directly make use of local variables:

```
string *online_creator_names( string who ) {
  object *creators = filter( users(), (: $1->query_name() == who :) );
  return map( creators, (: $1->query_name() :) );
}
```

This will not work — you'll get a compile time error telling you that it is "Illegal to use local variable in functional". If you want to use a variable like this, you need to explicitly force it to be a literal value rather than a variable by enclosing it in `$()`, like so:

```
object *creators = filter( users(), (: $1->query_name() == $(who) :) );
```

This syntax tells the MUD “hey, I’m 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 with `call_out()`. 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.

15.7. Conclusion

Powerful as all get out, but costly in terms of code readability and often efficiency — function pointers are the guilty little indulgences of the Discworld MUD. 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 our very first introductory text for new Discworld creators.

Many of us use function pointers 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.

16. Achieving Your Potential

16.1. Introduction

Achievements are a widely-used mechanism within the Discworld MUD, because they offer a way to get an extra degree of “oomph” out of our code without us 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 it 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, then, is largely a self-contained chapter on how to use our achievements system, with no reference to Betterville.

16.2. 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.

Achievements are rewards for participating and excelling in the game. They are publically announced (when they are above a certain level), and available for browsing. They have no requirement to be in-character (IC) — they can be a combination of IC and out-of-character (OOC) elements. They can be for achieving certain benchmarks in the game (e.g. hitting certain guild level targets), or for passing some threshold of participation (e.g. beheading 100,000 people).

Quests have a different purpose. We have always vaguely operated 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. 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 IC
- 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.

16.3. 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's a temptation to put in a dozen achievements for each minor development to make sure people pay enough attention, but alas, that just devalues the rest of them. Imagine, for example, adding a street with five shops, each with a "Buy $X of stuff from this shop" achievement. That's a pretty transparent attempt to curry interest. 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.

Each achievement gives the player a certain number of achievement points (AP), so we can think in terms of an "AP budget" to give us a rough guideline:

Size of development	Suggested 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 (e.g. Sto Lat)	10 AP
A moderate-sized subsystem	10 AP
A complex subsystem	15 AP
A small city (e.g. DJB)	20 AP

A medium-sized city (e.g. Genua)	40 AP
A large city (e.g. BP)	60 AP
A huge city (e.g. 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 might 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.

As an example of Forn back when Genua was put into the game, we’d be looking at the following budget:

Subsystem	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”.

16.4. 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 leader of that domain who has the authority for approving or rejecting achievements. For `/obj/achievements/`, you should get approval from a sufficiently-senior creator; that is, an Independent, Director, or Trustee.

Let’s take a look at a simple achievement. This one is from the Ram domain:

```
::item "ram:cutthroat twit"::
::->name:: "cutthroat twit"
::->story:: "failed to heed a warning"
::->level:: ACHIEVEMENT_MODERATE
::->criteria:: ([ "teeny sod haircuts" : 1 ])
```

```
::->category:: ({ "mortality", "quirky", "ramtops" })
::->instructions:: "Visit Teeny Sod in Ohulan-Cutash and have one of his "
                   "very, very special haircuts (or one of his very, very "
                   "special shaves, if you're sufficiently "
                   "follically-enabled)."
```

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*, which takes this file and converts it into a class file suitable for parsing by the achievements handler.

The first entry in this data file (the *item*) 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.

The *name* is the title your achievement is going to have — this is what will appear when people browse or achieve it.

The *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:

```
Cutthroat Twit, in which you failed to heed a warning.
```

You shouldn't include the text "in which you" — 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 `/include/achievements.h`. Minor achievements award 5,000 XP 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* — it's the most important part of your achievement file.

The *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.

16.5. Criteria

Okay, but what about that criteria part? That looked a bit more complicated!

The criteria is what drives your achievement — it 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:: ([ "teeny sod haircuts" : 1 ])
```

This achievement is granted to a player when that player's "teeny sod haircuts" criteria value is ≥1. 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. Your code tells the achievements handler to change this value, and the handler does the rest.

As an example of this in action, take a look at `/d/ram/chars/Ohulan-Cutash/teeny_sod.c`, an NPC who gives "haircuts" in a small Ramtops town. This file includes the following code:

```
#define ACHIEVEMENT_CRITERION "teeny sod haircuts"
[...]
void die_silly_person( object player ) {
  [...]
  // Kill them dead.
  tell_object( player, "Your life slips away...\n" );
  player->attack_by( this_object() );
  player->do_death();
  [...]
  ACHIEVEMENTS_HANDLER->adjust_player_achievement( player->query_name(),
    ACHIEVEMENT_CRITERION, 1 );
  [...]
}
```

So a player who lays down their life for their (pie-loving) friends will have their "teeny sod haircuts" criteria value incremented by 1. 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 the player 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 can 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.

16.6. 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:: ([
  "teeny sod haircuts" : 1,
  "teeny sod shaves" : 5,
])
```

With this as a criteria, the player must have a value of ≥1 for their “teeny sod haircuts” value, and ≥5 for their “teeny sod shaves” 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 values) 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:

```
::->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 instead use `add_to_achievement_array()` like so:

```
ACHIEVEMENTS_HANDLER->add_to_achievement_array( player->query_name(),
  "people eaten", "drakkos" );
```

Arrays in an achievement have a maximum size of 10 — 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 10 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 executed three times, the achievement array would still only contain one instance of the name “drakkos”.

You can force it to have duplicates by adding a fourth 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()` method:

```
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 to 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 contain 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 10 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.

16.7. 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. Most achievements should absolutely be as simple as indicated above.

Still, 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 '“teeny sod haircuts”), the achievements handler can work out which criteria values belong to which achievements. However, if a criteria is a function pointer, the achievements handler has no way of knowing what achievements can belong to it. If the function pointer references an external object (such as another handler), the achievements handler also 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 achievement 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 use either the player handler (which has a number of useful methods), or the method `call_function_on_player()` in the achievements handler — this will check whether the player is online, and will 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() - ACHIEVEMENTS_HANDLER->call_function_on_player( $1->name,
       ({ "query_character_start" }) ) )
      / (60 * 60 * 24) :)  : 180,
])
::->category:: ({"social", "age"})
::->instructions:: "This achievement is awarded upon having a player account "
  "of six months or older.  If you have refreshed, the time starts counting "
  "from the time of your most recent refresh."
```

All of this is stuff we've talked about before, with the exception of the criteria:

```
::->criteria:: ([
  (: (time() - ACHIEVEMENTS_HANDLER->call_function_on_player( $1->name,
       ({ "query_character_start" }) ) )
      / (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 via the achievement handler, not on the player themself), and then divides it by the number of seconds in a day. If the value that comes out of that function pointer is ≥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 usually you'll only need its `name` entry: `$1->name`.

You might find upon doing this that you get an error telling you to "use a cast to disambiguate". What this means is that your code is including another class (in addition to `player details`) that also has a `name` entry, and the driver is unhappy about not knowing which class it should use for `$1`. The solution is simple, if a little ugly — change

```
$1->name
```

to

```
((class player_entry)$1)->name
```

You're certainly not always going to get this error, but if you do, that's how to fix it.

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 third parameter here refers to whether or not these achievements are being granted by the `achievements legacy` command; if it is set to `1`, it will suppress all informs except those that go to the player. We don't want that to happen after a simple bank deposit — capitalists deserve to be exposed! — so we set it to `0`.

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's heart_beat() (although they don't get triggered every heart beat — that would be way too resource-demanding), for the wealth ones they go wherever the state of a bank account is altered, and so on.

16.8. Other Achievement Settings

Titles can go along with achievements, but these (with rare exceptions) should only go along with achievements of the `LARGE` level or higher. Check with your supervisor or the relevant domain leader 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:[‡‡‡‡‡‡‡‡‡‡]

```
::->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 should have been 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 ) :)
```

[‡‡‡‡‡‡‡‡‡‡] Don't just delete the achievement itself — that won't make as much of a difference as you'd like.

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.

16.9. 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 MUD-wide achievements, and the appropriate achievements subdirectory 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 playerbase.

16.10. Ach, 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 was last achieved. and how many times it's been achieved, and so on. If you're absolutely sure you want that achievement to go away, you can do the following call:

```
   > call clean_achievement( "my ach" ) /obj/handlers/achievements_handler
```

And bam, away it will go!

16.11. 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:

Level	AP	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[§§§§§§§§§§]
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 straightjacket, and is 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. Ask your supervisor for advice 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 and the time since your first login are both things that you accrue regardless of any effort. Despite the guideline being a decade for a Mythic achievement, the “Dawn of the World” achievement actually requires 20 years to have passed, because the time passes without you doing anything.

§§§§§§§§§§ Breaking 100 street lights might take an hour or so to do and be a Small achievement, while getting a common skill to level 150 might take much longer — but it’s something you’d be doing anyway, so that could also be Small.

16.12. Conclusion

Achievements excel at adding value to existing code with a minimum of creator effort. It's well worth spending a bit of time thinking about where you could enhance your developments — and developments that are already in game — by using achievements to draw players to their various features.

17. So Here We Are Again...

17.1. 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.

17.2. 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_command()`, and the various syntaxes and patterns that go with it. This opened up a world of interaction possibilities within our rooms and objects. It 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 autoload 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 MUD 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, but you're well placed now to learn about all of that stuff yourself. There is always more to learn, but now you not only know enough to code some very interesting objects, you also have the necessary background information to be able to understand new and interesting code that comes your way.

17.3. 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 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 if you want to discuss something before documenting it. 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.

17.4. 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 on you.

You can also learn a lot from more general programming resources. 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.

17.5. Conclusion

So long, farewell, adieu. This text has been a long time coming. The first edition of *LPC for Dummies 1* was written in October 2000 by Drakkos, who had only been a creator for about a year and a half at that point. People liked it, though, and there were constant requests for a sequel that would address the more complicated parts of LPC development. That sequel is now in your hands! (Or, more realistically, on your monitor/tablet/whatever.).

Drakkos's plans for *LPC for Dummies 2* changed a lot over the years, and the material itself has changed further since he handed it over to the Discworld MUD creator team. Really now it's one of three tightly related works, which together 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 (or onto the creator wiki) and you should consider *LPC for Dummies 1*, *Being A Better Creator*, and *LPC for Dummies 2* to be three separate sections of one larger book. We hope, in your travels, you find it of use to you.

18. Reader Exercises

18.1. 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 beyond what you've learned about during the course of *LPC For Dummies 1* and *2*.

Some of them are easier than others, but all of them involve a degree of practising for yourself what we have worked through in our example development.

18.2. Exercises

18.2.1. Someone Forgot Their Code

As we noted earlier (see footnote on page 60), there is currently no way for a player to re-acquire their panel code if they forget it. Design and implement such a way.

18.2.2. Through the Ceiling Hole

We mentioned when adding our second quest that we'll need to add a command to allow the player to actually climb up through the hole in the library ceiling. So that they don't need to keep breaking a new hole every time they come back, it would be good to just make this conditional on them having completed the first quest. (However, since these quests are set to inactive so they don't show up in player scores, we'll need to comment that bit out to test it.) Write some code to do this.

18.2.3. Portraits in the Library

We also noted earlier (see footnote on page 77) that we haven't provided any way for players to actually look *at* the individual portraits; moreover, although we've allowed players to look behind individual portraits, we haven't given them a hint that they should do so (see footnote on page 78). Implement some add_items to fix both of these issues.

18.2.4. 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.

18.2.5. Researching Players in the 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.

18.2.6. Researching Other Concepts

Our library should also have information about concepts, so as to provide a narrative framework. In addition to allowing research into 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.

18.2.7. 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 skillcheck.

18.2.8. 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.

18.2.9. 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 `adventuring.perception` at the same time.

Also, we never did dress the young romantics, so finish off the stock for the shop and incorporate it randomly yet appropriately into the setup of the NPCs.

18.2.10. 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 it knows about, and all of the other concepts that should be represented.

18.2.11. Research Skill Check

It’s not easy to research in a library — certainly not something you always do and always get right. Incorporate skillchecks into the library so that sometimes you don’t find anything, and sometimes you find misleading information.

18.2.12. Helpfiles for the 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 helpfile for the library that details the syntax and functionality available. Then add it to the library so that `help here` functions correctly.

18.2.13. 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.

18.2.14. Rub The Pain Away

Players who rub themselves with dock leaves will only be notified that this has had any effect on their nettle sting if the result of the rubbing was enough to completely remove the effect. Amend the code so that if it *wasn’t* enough for complete removal, they get a message telling them that the intensity of the effect has decreased.

18.3. 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 your ideas to the leader or deputy of the Developers domain, and we can look at incorporating them into this section!