LPC For Dummies

Book One

Michael Heron
(drakkos@discworld.atuin.net)

Beta Draft

Table of Contents

Mojo The Monkey Says..5
Willkommen! Bien Venue! Welcome! 6
  Introduction 6
  The Learning Scenario..6
  The Structure of the Game 7
  Programming as a Concept 8
  It's All Down To You..11
  Conclusion..12
My First Room..13
  Introduction 13
  Your Own Learnville..13
  The Basic Template..14
  Now, what does all of that mean? 15
  My Room Sucks..16
  It's a little too quiet..17
  A Second Room 18
  Our Overall Plan 19
  Property Ladder 20
  Conclusion 21
My First Area..22
  Introduction 22
  The Structure 22
  Exits and Maintainability 24
  Our Exitses 26
  Chain, chain, chain... 29
  Conclusion 31
Building The Perfect Beast..32
  Introduction 32
  A First NPC 32
  Breathing Life Into The Lifeless 34
  Cover Yourself Up, You'll Catch A Cold 37
  Request Item..40
  Chatting Away 41
  Conclusion 42
Hooking Up..43
  Introduction 43
  The Path.h Problem 43
  Sorted!..45
  If At First You Don't Succeed 47
  Compound Interest 50
  Moving 51
  One Final Touch 53
  Conclusion 53
Back To The Beginning..55
  Introduction 55
  Captain Beefy's Return 55

The Road Less Travelled 56
Bigger, Better, Faster 58
Probing Dark Depths 59
The Taskmaster 60
Switching Things Around 62
Scoping Things Out 63
Conclusion 65
Now That We're An Item..66
Introduction 66
Virtually Self-Explanatory 66
But what does it all mean? 69
Beefy's Boots 70
Bling 71
A Word Of Warning About Items 73
Conclusion 73
An Inside Job..74
Introduction 74
The Mysterious Room 74
Modifying Exits 75
Shop 'Till You Drop 78
Stabby Joe 80
Conclusion 81
Dysfunctional Behaviour..82
Introduction 82
Our Task 82
The Science Bit... Concentrate! 82
The Structure of a Function 84
A Little Bit More... 86
Function Scope 87
Onwards and Upwards!..88
Violence Begets Violence 91
A Local Shop For Local People 94
Conclusion 95
Going Loopy..96
Introduction 96
A Basic Pub 96
Grumpy Al 97
Grumpy Al Goes Ballistic 100
Components 101
Loops 104
Doing While... 105
For Loops..106
Fire in the Hole! 108
Conclusion 109
Arrays, You say?..110
Introduction 110
Slicey Pete 110
The Array 113
Array Indexing 115

Array management........116
Setting Up Stock, LPC Style........117
The Foreach Structure........119
Conclusion........119
Hooray for Arrays........121
Introduction........121
A Secret To Be Discovered........121
Item Matching........123
A Grue Some Fate!........126
Getting To Our Shop Of Horror........130
Conclusion........131
Mapping It Out........133
Introduction........133
The Mapping........133
The Magic Hate Ball........136
More Mapping Manipulation, Matey........140
Conclusion........142
So Long........143
Introduction........143
The Past........143
The Present........145
The Future........146
Conclusion........146
Reader Exercises........147
Introduction........147
Exercises........147
Send Suggestions........149

Mojo The Monkey Says...

All rights, including copyright, in the content of these documents are owned or controlled by the indicated author.

You are permitted to use this material for your own personal, non-commercial use. This material may be used, adapted, modified, and distributed by the administration of Discworld MUD (http://discworld.atuin.net – try the veal) as necessary.

You are not otherwise permitted to copy, distribute, download, transmit, show in public, adapt or change in any way the content of these web pages for any purpose whatsoever without the prior written permission of the indicated author(s).

If you wish to use this material for non-personal use, please contact the authors of the texts for permission.

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

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

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

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

Willkommen! Bien Venue! Welcome!

Introduction

This is the Discworld – flat, circular, and carried through space on the back of four elephants who stand on the back of Great A'tuin, the only turtle ever to feature on the Hertzsprung-Russell Diagram, a turtle ten thousand miles long, dusted with the frost of dead comets, meteor-pocked, albedo-eyed. No-one knows the reason for all this, but it's probably quantum.

Much that is weird could happen on a world on the back of a turtle like that...

... and it's now your job to ensure that it does!

In this set of introductory creator material, we are going to talk about the way in which the gameworld of Discworld MUD is constructed. This is a book about coding, but don't worry about it - we're going to introduce the topic as gently as we possibly can. New topics will be introduced only when they help us make our code better and more interesting. You will find all of the source code that goes along with this text available in the /d/learning/learnville/ directory. It is broken up into chapters, so you can see exactly what your code should look like at each stage of development. You can goto the room 'access_point' within this directory to explore all of these versions and see the differences between them.

The Learning Scenario

Over the course of this text, we'll be building a game area – exactly like the kind you have experienced in the game. We're going to create rooms, NPCs, items, and shops. We're going to stop short of adding unique functionality or quests, because that's a part of the follow-up creator text. By the end of this text you should be comfortable with the basic building blocks of LPC. Along the way you'll even learn a bit about code, but only a bit. Being a Discworld creator involves working with code, but you don't need to be a coder to be a creator. Enough to get by will take you far, although it's always good to learn more if you can.

Our learning scenario is the development of the village of Learnville. A fully completed version of this may be found in the learning domain under /d/learning/learnville/chapter_13. I would like you though to construct this step by step in time with the teaching material – you only get a fraction of the benefit from reading completed code that you do from writing the code as you go along.

Learnville is a simple village, with basic facilities. It also doesn't contain much in the way of descriptions - if you want to see what good descriptions look like, take a look at some of your favourite game areas. Our scenario is for teaching you how to put an area together, it is assumed that you'll be able to write the text yourself. Some guidance on the topic of writing descriptions may be found in Being A Better Creator. What we're interested in here is the structure of the area, not how it looks.

We're going to approach this functionality according to a development technique called incremental development. I heartily recommend this for beginners because it greatly simplifies the job of building complex programming structures. This is a process of beginning with something very simple until you have the very basic framework fitting together. Once you have the skeleton, you can start adding features to it a little bit at a time. In this way, you avoid becoming bogged down in the overwhelming complexity of a project – you get to see it grow all around you from the most humble beginnings to a complete and fully featured area. To put this in context, we start with a single room that does nothing exciting. We then add a second room and connect these two skeleton rooms together. Then we add in another, and another, until we have a whole framework of a village ready for us to fill in the blanks. What we don't do is create the whole framework to start with and then become flummoxed when it doesn't work.

Incremental development is an exciting and satisfying process, and one with which you are going to become entirely familiar as you work your way through this material. You can see your developments shaping up in a very tangible way, and that invites further development. Trying to do everything to start with can be a soul-crushing way of developing a project.

The Structure of the Game

Everything on Discworld MUD is something called an object. This is quite a complex topic in itself, so we're only going to scratch the surface of this for now. Every room is an object, every item is an object, every player is an object. Confusingly, object doesn't mean that it's necessarily something you can touch or interact with in the game. Your guild commands for example are also objects. It's not an easy thing to get into your mind.

In programming terms, an object is a structure that contains some data, and instructions for acting on that data. Some objects also take on the properties of other objects through a mechanism called inheritance. This is the same general biological principle you'll be familiar with in real life, only cast in programming terms. In real life, a cat inherits the properties of being a mammal, which in turn inherits the properties of being an animal. We'll see this mechanism in action from the very first code we write - our object inherits the properties of being a room.

Most of the code you need to work with on Discworld has already been written for you – you just need to tell the MUD what things you want to happen, and when. Most of the things you'll want to do, at least in the short term, are available for you. When you want to get a little more adventurous, then you'll be looking at moving onto the more advanced creator material.

The MUD itself is built on three separate structures. The first of these is the driver, which is known as FluffOS. You won't need to worry about that just now. The second part is the mudlib, and it defines all of the code on which the rest of the game - the domain code - is built. It's domain code that you will be building in the short term, and that's all the code you'll find underneath the /d/ directory. The mudlib is everything else, including code that will Scar and Terrify you if you read it too early.

As a basic guide to the relationship between the mudlib and the driver, it can be summed up like this – the mudlib knows what to do, but doesn't know how to do it, and the driver knows how to do it, but doesn't know what to do.

Domain code includes all of the areas in which you adventure as a player, all the quests you have encountered (and the ones you haven't), and all the NPCs you've slain.

When an object is made available in the game, it must first be loaded. At that point, it becomes a master object. Many objects remain as master objects, which means only one of them is ever in place in the game – rooms and unique NPCs are good examples of these.

Some objects are clones, and that means they are copies of the master object. Generic, common or garden NPCs and items are examples of cloned objects. When we need a new pirate for example, we just take a copy of the master object and put that copy in the game.

In order for an object to load, we use the update command. If our code is all Present and Correct, the object will load. We'll look at this more in the upcoming material. If an object has been incorrectly coded, you will be presented with lines of red text (syntax errors) and blue text (warnings). We call these 'compile time' errors - every syntax error in your code must be fixed before an object will load, but it will load if warnings are present. Fix them anyway. The process of writing code, updating an object, and then fixing the syntax errors will become second nature to you before too long.

Programming as a Concept

Programming is a difficult task, unlikely almost anything you may have learned before – it's like learning how to do mathematics in a foreign language – a confusingly new syntax combined with a formally exact vocabulary. When you learn how to speak a different language, you can rely on your interlocutor to be able to divine meaning even when you make mistakes – a computer cannot do that. And worse, it does exactly what you tell it to do – that sounds great in theory, but it can be hugely problematic in reality.

Computers don't understand much – at their basest levels, they are machines for manipulating electrical impulses. At the lowest level, these are interpreted by the computers as a collection of ones and zeroes (a numerical system called **binary**, but you don't need to know anything about that just yet).

However, we don't want to write our programs as ones and zeroes – it would take a long time, require a lot of tedious busy work, and generally be a massive drag. The first programmers however did exactly that, they wrote their programs in the language of the computer itself - **machine code**.

As time has gone by, it has become simpler to write programs for computers – in order to manage the complexity of programming as a task, computer people developed successively more abstracted programming languages, and introduced the idea of **compilation**. Rather than writing in machine code, programmers could write in an easier, simpler system and then make the computer itself convert that into machine code. This conversion process is called **compilation**.

The first attempts to do this were through a language called assembler, which is only a little bit better than machine code directly. Assembler is known as a **low level language** because it's not all that far removed from the low level grunts and snorts of machine code.

As the years have gone on, more and more programming languages were developed. Successively they abstracted writing programs into more and more English-like syntax. Eventually a language named C came along, and revolutionized how people thought about programs. C was so successful that most of the successful programming languages of modern times use its style of writing code in one way or another – C#, C++ and Java especially.

These are known as **high level languages** because of the level of abstraction involved. Computers do not understand the code written in these languages, but the process of compilation converts the code into something the computer can understand.

On Discworld, we use a language called **LPC** which was created by Lars Pensjoe. While it has very similar syntax to all of the above named languages, it has a host of unusual and useful features. We'll get to what those features are as we go through the material.

Programming is essentially the task of taking a list of instructions, and writing them in such a way that the computer can follow your instructions to complete a task. To give a simple, real world example – imagine the task of making scrambled eggs. Imagine you have an obedient, but simple-minded kitchen assistant, and you would like them to make some scrambled eggs for you. They do everything you tell them to do, and they do it exactly as you tell them. It sounds like an ideal situation, but consider the following instructions:

1. Get two eggs from the fridge
2. Get some milk from the fridge

3. Get some butter from the fridge
4. Break the eggs into a jug
5. Whisk the eggs until they are well mixed.
6. Heat some milk and some butter in a pot.
7. Add the egg mixture.
8. Stir until scrambled eggs are made.

The instructions are pretty simple, but your Simple Minded Assistant falters at the first instruction. He, or she, stands vacantly at the fridge, pawing ineffectually at its surface. Cursing, you amend your instructions a little:

1. **Open the fridge, you insensitive clod.**
2. Get two eggs from the fridge
3. Get some milk from the fridge
4. Get some butter from the fridge
5. *Close the fridge*
6. Break the eggs into a jug
7. Whisk the eggs until they are well mixed.
8. Heat some milk and some butter in a pot.
9. Add the egg mixture.
10. Stir until scrambled eggs are made.

You set your Simple Minded Assistant back to work – they open the fridge. They get two eggs out... success! Then they stand there with a confused look on their face. "What does 'some' mean?", they ask. You curse once more:

1. Open the fridge, you insensitive clod.
2. Get two eggs from the fridge
3. Get **two tablespoons of** milk from the fridge
4. Get a **knob of** butter from the fridge
5. Close the fridge.
6. Break the eggs into a jug
7. Whisk the eggs until they are well mixed.
8. Heat some milk and some butter in a pot.

9. Add the egg mixture.
10. Stir until scrambled eggs are made.

And then you start them on the task. "Uh, the fridge is already open", they say, and then the metaphysical complexity of the situation causes them to shut down until you ammend the program a little further:

1. **If the fridge door is shut**, open the fridge, you insensitive clod.
2. Get two eggs from the fridge
3. Get two tablespoons of milk from the fridge
4. Get a knob of butter from the fridge
5. Close the fridge.
6. Break the eggs into a jug
7. Whisk the eggs until they are well mixed.
8. Heat some milk and some butter in a pot.
9. Add the egg mixture.
10. Stir until scrambled eggs are made.

And so on, until you have a completely correct, entirely unambiguous list of instructions that any dolt could follow. This is the essence of what programming is all about. The only difference is, your instructions are given in LPC, and the Simple Minded Assistant is the MUD itself. Trust me, it's even more stupid than your kitchen assistant.

This is a much trickier task that I'm making out here – and the only thing that makes it easier is practice - **lots and lots** of practice. That leads us to the next section of this chapter...

It's All Down To You

Whether you are learning to code on Discworld, or in a more formal setting like a university course, the basic proviso is the same – **nobody can teach you how to code**. Sure, people can point you in the right direction and provide you with ample reading and reference material. What they can't do is make you understand, because you don't learn it by having someone explain it to you. The only way you learn how to code is by practicing.

But more than that – the only way you really learn is by trying and making mistakes. If you get it right the first time, that's great – but you would have learned more by getting it wrong and working out what why your code didn't function the way you hoped. I guess what I'm saying is – don't get discouraged. You are going to make mistakes, and sometimes you're not going to know how to fix them. That's what the **Mentats** in the Learning Domain are there to help you with – but please do try to work it out first before you ask for help. Not because they don't want to help you, because they do – but because you'll learn more if you work out the answer yourself.

You've got to be willing to persevere in this – the easiest way to fail is to not try. We can support you in your crusade to be a Good Creator, but only if you meet us half way. Remember, your domain lord thinks you can make it as a creator, or you wouldn't have been hired in the first place.

Conclusion

In this chapter we've gone over the very, very basics of what programming actually involves, and how code on Discworld is put together. All of this is useful knowledge, but it's not coding yet. That comes in the next chapter when we look at building our very first room. Are you excited? Touch your nose if you are!

My First Room

Introduction

Right, let's get started for real then – we've got a lot to learn and you've already read so much without getting your hands on a bit of code. Now we're going to change that by creating our very first room and filling it with some of the simpler elements available to a novice creator. This is a big step - even the largest of our area domains started off at one point with a single room.

Once we've gotten that first achievement under our belt, we'll talk a little bit about structuring an area so that we don't get hopelessly bogged down in the minutiae. We're going to look at the basic skeleton of Learnville, and how many rooms we're going to develop – having a plan always makes it easier to schedule your time and effort. You should always sit down and sketch out what you are hoping to accomplish with an area before you start writing any code. The easiest time to make corrections is at the conception stage.

Your Own Learnville

The assumption throughout this material is that you are going to be building this village alongside the tutorials. In order to link up what's done in these documents to what you're doing on the MUD, we need to calibrate! That's just a fancy-pants way of saying that we're all using the same directory names.

First, you create a directory in your /w/ drive called learnville. All subsequent directories will be created in here. You don't need one for each chapter as is present in /d/learning/learnville - you should have one copy of this that grows and changes as you develop it further.

```
mkdir /w/your_name_here/learnville
```

Once you've gotten the directory, you want a subdirectory in there called *rooms*:

```
mkdir /w/your_name_here/learnville/rooms/
```

From this point on, this chapter will assume this is where you are uploading your files. Make sure it is, because otherwise you're going to have lots of problems ensuring that the paths you use map up to the paths we use in here.

The Basic Template

First of all, let's set up our basic template for creating a room. Create a new file in your development environment (See Welcome To The Creatorbase for a discussion on that if you haven't already done this), and type the following into your editor:

```
inherit "/std/room/outside";

void  setup() {
  set_short ("simple village road");
  add_property ("determinate", "a ");
  set_long ("This is a simple road, leading to a simple village.  There "
    "is very little of an excitement about it, except that it represents "
    "your first steps to becoming a creator!\n");
  set_light (100);
}
```

Save this file in your /rooms/ subdirectory, and call it street_01.c.

Now, in the last chapter we spoke a bit about loading objects – this room doesn't yet exist on the MUD, even though it exists on the disk-drive. To see if it loads, we use the update command:

```
update /w/your_name_here/learnville/rooms/street_01.c
```

When you hit return, you should see a message something like this:

```
Updated simple village road (/w/your_name_here/learnville/rooms/street_01)
```

If it doesn't, then something has gone wrong with what you've typed into the editor. Look very closely at the example code above and see where your code differs – it should be identical. If all else fails, simply copy and paste the code above into your editor - don't get bogged down in the errors just yet. You'll get your fair share of those later when you try to do things without a safety net.

Once your room has loaded, you need to move to it with the goto command:

```
goto /w/your_name_here/learnville/rooms/street_01
```

If all has gone well, you'll see the following:

```
/w/your_name_here/learnville/rooms/street_01 (unset)
It is night and there is no moon.
  This is a simple road, leading to a simple village.  There is very
  little of any excitement about it, except that it represents your first
  steps to becoming a creator!
It is a freezing cold backspindlewinter's night with a strong breeze, thick
black clouds and heavy snow.
```

```
There are no obvious exits.
```

Congratulations! You've just taken your first step into a larger world!

Now, what does all of that mean?

Let's take a look at what you've done and explain what each of the different bits mean. Most of them should be fairly obvious, but it's important that you know the bits you don't need to worry about:

```
// This the object that defines all of the functionality of which we're
// going to make use.  All of the code that we use in setup is defined in
// this object.  In this case, it's an outside room... so it gets all the
// usual weather strings and chats.

inherit "/std/room/outside";
// This is a function (we'll come back to this topic later).  All you need
// to know for now that the code in setup gets executed automatically by
// the MUD when your room is loaded.

void setup() {

  // The short description is what you see when you 'glance' in a room.
  set_short ("simple village road");
  // The determinate is what gets appended to the short description
  // sometimes.  In this case, this room becomes 'a simple village road'.
  // If we wanted it to be 'the simple village road', we'd change
  // the determinate here to 'the'.
  add_property ("determinate", "a ");
  // This is the long, wordy description of the room that you see when you
  // have the MUD output on verbose.
  set_long ("This is a simple road, leading to a simple village.  There "
    "is very little of an excitement about it, except that it represents "
    "your first steps to becoming a creator!\n");
  // This sets the current light level for the room.  100 equates
  // to 'bright sunlight'
  set_light (100);
}
```

The lines that start with the // symbol are comments. The MUD completely ignores these when it is loading your room – they are there purely for humans (and other creators) to read. It's good practice to comment your code properly, but that's a topic for a later day. You'll read more about commenting in Working With Others.

At the simplest level, all of the code that you incorporate into your rooms goes into the setup of a room. You need to pay particular attention here to the curly braces – it's easy to get these wrong when you first start – all of your code goes between the opening curly brace **{**, and the closing curly brace **}**.

That's the basic structure of each room we're going to create – you may want to copy and paste this somewhere you have easy access to so you can use it for each room you start.

My Room Sucks

Yes, yes it does! Your room is dull, boring, uninteresting… generally lame and horrible. But it loads – let's take one step at a time!

Let's look at adding some more interesting things to your room. The first thing you'll notice is that you can't look at anything… 'look village' gives a rather disheartening response:

```
Cannot find "village", no match.
```

Let's start off simply and add some things for people to look at. We do this by using add_item:

```
add_item ("village", "The village is simple, but beautiful because of it.");
```

This goes into the setup of your room – as a matter of convention, it goes after you've set the long description for the room. Update your room, and 'look village'. Your response will be much more encouraging:

```
The village is simple, but beautiful because of it.
```

Looking at the road will give us the 'no match' message, so let's add an item for that too:

```
add_item ("road", "You're standing on it, you dolt!");
```

You can add as many items as you feel is appropriate. In general, more is better – but at the very least every noun in the main description, and every noun in the add_item descriptions should have an add_item of its own.

But wait! We're not quite safe here, because the long description refers to a 'simple village', and what happens when we look at the simple village? Yep, 'no match'.

We get around that by being more specific in our add_item definitions:

```
add_item ("simple village", "The village is simple, but beautiful because"
  "of it.");
add_item ("simple road", "You're standing on it, you dolt!");
```

Add_item is quite clever – by doing this, we get the root noun (village) plus any adjectives we define, in any order in which we want them... looking at 'village' gives us the description, as does look at 'simple village'. If we had an add_item for 'shiny red simple village', we'd be able to look at 'shiny village', 'red village', 'shiny simple village' and any combination thereof.

We'll return to add_item intermittently as we go along – it's much more powerful than a lot of people realize, and we'll have good cause to make use of that power in later chapters.

It's a little too quiet...

Most rooms in the game have a little more life to them than your first attempt. They have NPCs (we'll get to those), and they have chats that appear at semi-random intervals. These chats are a big part of making a room feel alive, so let's add some of those.

Again, in your setup function, add the following code. It's slightly complicated, so you may want to copy and paste it:

```
room_chat ( ({ 120, 240,
  ({
    "It's quiet... a little too quiet.",
    "It's simple... a little too simple.",
    "It's exciting learning to develop rooms!",
  })
}));
```

Yikes! What's with all the brackets and braces? Let's pretend we don't need to worry about that just now, because we don't – it'll all come clear in later documents.

Instead, we shall focus on the Salient Details here – the first are the two numbers. These are pretty straightforward – the first is the minimum time that must pass before a chat is sent to the room, and the second is the longest period of time that will pass before a chat.

The strings of text are the actual chats that get echoed to the room – you can have as many of these you like, inserted into the list. Each needs to be separated by a comma:

```
room_chat ( ({ 120, 240,
  ({
    "It's quiet... a little too quiet.",
    "It's simple... a little too simple.",
    "This is a chat in a list!",
    "It's exciting learning to develop rooms!",
 })
}));
```

Note the quotation marks – we'll talk about why these are necessary in a later chapter of the material. Just take my word for it now that they're needed.

The only thing missing from our most basic of rooms is an exit to another room. For that, we actually need a second room to move to!

A Second Room

Let's start with out basic template as before, changing the long description a little to reflect the fact this is somewhere different. We'll call this one street_02.c, so create a new file and save it to the MUD:

```
inherit "/std/room/outside";

void setup() {
  set_short ("simple village road");
  add_property ("determinate", "a ");
  set_long ("This is a little way along the path to the village of "
   "Learnville.  The path continues a little towards the northeast "
 "towards the rickety buildings of the market square.  To the "
 "southwest, the path wends away from what passes as civilization "
 "in this learning scenario.\n");
 set_light (100);
}
```

We now want to link these two rooms up – this is done using add_exit. Add_exit requires three pieces of information before it can make up a connection to a room – the first of these is in what direction the exit lies. The second is what path the next room may be found at. The third is the type of exit (is it a door, a path, or stairs?). Our long description above shows us what our direction is going to be – our first room lies off to the southwest of this room. It's going to be an exit of type 'path':

```
add_exit ("southwest", "/w/your_name_here/learnville/rooms/street_01",
"path");
```

This goes into street_02.c. Street_01.c gets its own exit leading to our second room:

```
add_exit ("northeast", "/w/your_name_here/learnville/rooms/street_02",
"path");
```

Update both of these rooms, and you'll be able to walk between them. That is pretty nifty, I'm sure you'll agree!

One thing to bear in mind is that while we've put the full path of these rooms in our add_exits, that's not how we should do it 'In Anger' because it makes it really difficult to move areas around as you need them. Later, we'll talk about the importance of properly structuring the architecture of your code so that it's easy to move things around various directories.

You'll note that our second room is missing add_items and room_chats – adding these is left as an exercise for the interested reader. You should be able to do that yourself referring to the notes for how it is done in the first room.

Our Overall Plan

So, that's two simple rooms linked up. It's worth making sure we have an actual plan here so that we know how each room fits into the whole. You may have your own favourite mechanisms for designing a map for a new area, I personally favour the humble text-file. This is the map of the area we're going to develop together:

```
          A  B
          |  |
          6-7-C
          |  |
        3-4-5-D
       /
      2
     /
    1
```

We're using a number here for each outside room, and a letter for each inside room. We thus need a key so we know which number is which room:

Number	Filename
1	street_01.c
2	street_02.c
3	street_03.c
4	market_southwest.c
5	market_southeast.c
6	market_northwest.c
7	market_northeast.c

We'll worry about the inside rooms later. It doesn't matter particularly what filename we give the rooms, but to ensure that your work and these tutorials sync up, you should be making use of the same filenames as we use. Trust me, it's just simpler that way, for now.

A text file is ideal because it can be viewed on the MUD, or put easily into the wiki. Here, I've put a number for each room we're going to create, and letters for inside rooms with Further Features. It's a small area, but one that will cover everything we need to discuss in order for you to make a meaningful contribution to your domain. We've done room 1 and room 2. In the next section we'll add in the skeleton of the remaining rooms as we encounter new things we can develop.

Property Ladder

As part of our code above, we have a line that references to 'add_property'. Properties are little tags that can be added to an object whenever they are needed, but they don't actually do anything other than exist. We can check in other pieces of code whether an object has a property attached, and if so change the way we treat it a little.

As an example of this working, you can add the 'gills' property to yourself to give you the ability to breathe under-water:

```
call add_property ("gills", 1) me
```

The first part of this is the property to add, and the second is the **value** to give that property. The 1 simply indicates 'yes, this property should be added'. Now, you'll never drown when you're in the water.

Properties are used intermittently through the MUD, but you should be wary of them - they have a tendency to clutter objects. Because they are so easy for people to add (there is no set list of properties which are valid), it's tremendously easy for creators to write objects that attach properties to players and items, and then never write down why or what the property is for. Before too long, everyone is wandering around with properties like "gnah bag check", and no-one except the original creator (who may have left by this point) has any idea where it came from.

Properties are a quick and easy hack, but you shouldn't use them too much. In circumstances where you do, you should always make sure that they are **timed** properties. You can do this by providing a third piece of information to the add_property call - the number of seconds the property should exist for:

```
call add_property ("gills", 1, 120) me
```

Here, the gills property will remain on you until 120 seconds have passed, at which point it won't. These are 'self-cleaning' properties and you should always, always use them unless you are very sure indeed that you don't need to.

Conclusion

These simple rooms aren't very interesting – we're going to change that as we go along. We have taken a big step though – creating an actual real room you can stand in, and another room you can move to - it's from these basic building blocks that whole cities are built. We're going to leave them lacking in add_items and long descriptions and such - nothing is gained by writing these rooms to the standard that would be expected in the game. Once you've seen one add_item, you've seen them all - you don't learn more by having something repeated over and over again. Filling in the blanks is left as an exercise for the interested reader - I heartily encourage experimentation to see what happens when you change things - the fact that you have easy access to the original source-code from /d/learning/learnville means that there is no cost if you break everything utterly.

My First Area

Introduction

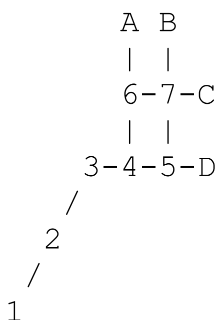
You've taken your first steps to developing a small room within the MUD. In this chapter, we're going to extend this to an entire area. In the process, we'll talk a little bit about how to develop an architecture that means we can easily move things around as we need, and link them up with minimum fuss. This is an important aspect of developing an area – things get moved around a lot on Discworld, especially through development and playtesting. Areas that are difficult to maintain are a drain on everyone's time. Luckily your areas aren't going to do that, because we're going to talk about how you can avoid it.

Once we've looked at the setup of the skeleton of the area, we're going to spend a little more time looking at the exits we've setup. The more detail we put into our areas, the richer our game world seems. That's a goal I'm sure we can all agree on being worthwhile.

The Structure

You've seen the map from which we're going to be developing. We can develop room by room, only adding a room into the whole when it's completed. However, in order to get a feel for how things are going to link up, it's beneficial to put skeleton implementations of each room in place, and then incrementally add their descriptions and features. That's what we're going to do now. It's easier than you may think.

Remember our map:



And our key:

Number	Filename
1	street_01.c
2	street_02.c
3	street_03.c
4	market_southwest.c
5	market_southeast.c
6	market_northwest.c
7	market_northeast.c

We've got the basic outline of rooms 1 and 2. We have another five rooms that will be outside rooms, and then four that will be inside rooms. We begin putting this framework together with a template. Save it as street_03.c:

```
inherit "/std/room/outside";

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

This is the room you're going to repeat for each of our outside rooms - there's nothing in it, but that's fine - we don't want to spend time writing some intricate, beautiful descriptions we are only going to have to scrap later. The four remaining rooms of our development are the market rooms. Now comes the magic!

You don't need to painstakingly create a new file for each of these rooms and then save it – we make use of the cp (copy) command to create the rest of the template rooms:

```
cp street_03.c market_southwest.c
cp street_03.c market_southeast.c
cp street_03.c market_northwest.c
cp street_03.c market_northeast.c
```

Now when you ls (list) the directory, you'll find it full of your skeleton rooms.

```
1 market_northeast.c    1 market_southwest.c      1 street_03.c*
1 market_northwest.c    3 street_01.c*            1 market_southeast.c
1 street_02.c
```

We haven't added the exits to these new rooms, but that's okay – we need to talk a little bit about exits before we leap into that part.

Exits and Maintainability

If you've been looking through the 'here's one that I made earlier' versions of these files in the /d/learning/learnville directory, you'll see that it's broken up by chapter. Each directory contains the code as it stands by the end of the appropriate chapter. I'm not writing this code every time – I'm copying it across from the last version.

However, the exits from chapter_02 don't work after moving it into the directory of chapter_03 – every time a move is made, you end up in the room for chapter_02. You can change the filenames by hand, and that's not too bad for two rooms. We've got seven now, and four more to come – it would be a nightmare to have to change all of those paths by hand, for every chapter of this book. It's too much to ask of anyone, especially me because I really don't want to have to do it.

The same thing is true with your code – sometimes things get moved around as directories are reshuffled, cities are redesigned, and generally treated with an incredible amount of casual disrespect. Even the Gods don't pick up cities and rehouse them with the indifference we do. Changing every exit in a large city by hand would be an impossibly annoying task, and one you'd just need to repeat when the city moved directories in the future. It's not a sustainable approach.

We have a way around that problem though – we make use of what is known as a **header** file.

Create a new file and save it as path.h (note, .h on the end rather than .c). It's going to contain one single line:

```
#define      PATH         "/w/your_name_here/learnville/rooms"
```

This is a special kind of code statement in LPC. Technically it's called a **preprocessor directive** – it's something that LPC deals with before it ever gets to your code. There is a subsystem in the driver called the **preprocessor**, and it's kind of a souped-up search and replace tool. For the code we have here, we have given the preprocessor a directive to **define** the text PATH as being another string entirely.

When you create your path.h file, make sure you hit return at the end of the line. If you don't, you'll get a worrying error saying something about #pragma strict_types. If this happens, go back to your path.h file and add a return at the end.

The preprocessor is essentially a sophisticated search and replace routine – providing a #define statement tells the preprocessor that every time it sees the first term to replace it with the second term. Thus, any time you use the text PATH in your program, it will replace it with the text "/w/your_name_here/learnville". This happens transparently, and every time the object is loaded. You see no impact on your code.

Simply creating a path.h file is not enough though, we need to tell our rooms to use it. We do this by using another directive called an **include** directive. At the top of each of your room files, before your inherit line, add the following:

```
#include "path.h"
```

This tells the preprocessor to take the path.h file you just created, and incorporate it into your rooms – as such, each of your rooms gets the define that is set. In your skeleton rooms, the code will now look like this:

```
#include "path.h"

inherit "/std/room/outside";

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

Now, let's see what impact that has on our original two rooms. Remember we have already set up their exits, like so. In street_01.c, we have:

```
add_exit ("northeast", "/w/your_name_here/learnville/street_02", "path");
```

In street_02.c, we have:

```
add_exit ("southwest", "/w/your_name_here/learnville/street_01", "path");
```

We're going to change these so that they read as follows. For street_01.c, we will have:

```
add_exit ("northeast", PATH + "street_02", "path");
```

For street_02.c, we will have:

```
add_exit ("southwest", PATH + "street_01", "path");
```

Now, update your two rooms – you'll see you can move between them just as easily as you were able to do before. That's because before LPC even begins to try and load your code, the preprocessor looks through your code for any instance of PATH and replaces it with your define – so what LPC sees when it comes to load your room is the following:

```
add_exit ("northeast", "/w/your_name_here/learning" + "street_02", "path");
```

LPC is perfectly capable of adding two strings together, and once it's done that it ends up with the full path to the room. The difference here though is that if you change it in your path.h file, it changes in every file that makes use of that path.h – one change in the right place and an entire city can be shifted from one directory to another without anyone lifting a finger. It's very powerful, and an approach to development into which you should definitely get into the habit. Check with your domain administration to see how this gets handled in your own domain. Some domains have a single .h file (such as forn.h) that serves as a master file for every room. Some have a more distributed approach whereby smaller .h files are available to subsections within a development.

Our Exitses

Now, we need to go back over our skeleton rooms and add in the exits – you should be able to do this quite easily by yourself now, as long as you know which rooms are heading where (and the map will tell you that). Don't add any exits for the inside rooms yet – we'll get to those later.

To make sure your exits look the way they are supposed to, here's the code for the exits in each of your rooms:

street_01

```
add_exit ("northeast", PATH + "street_02", "path");
```

street_02

```
add_exit ("northeast", PATH + "street_03", "path");  add_exit ("southwest", 
PATH + "street_01", "path");
```

street_03

```
add_exit ("east", PATH + "market_southwest", "path");  add_exit ("southwest", 
PATH + "street_02", "path");
```

market_northeast

```
add_exit ("south", PATH + "market_southeast", "path");  add_exit ("west", 
PATH + "market_northwest", "path");
```

market_northwest

```
add_exit ("south", PATH + "market_southwest", "path");  add_exit ("east", 
PATH + "market_northeast", "path");
```

market_southeast

```
add_exit ("north", PATH + "market_northeast", "path");  add_exit ("west", 
PATH + "market_southwest", "path");
```

market_southwest

```
add_exit ("north", PATH + "market_northwest", "path");  add_exit ("east", 
PATH + "market_southeast", "path");  add_exit ("west", PATH + "street_03", 
"path");
```

There's a particular convention that is followed when adding multiple exits – when you walk around the game, the exits are presented to you in the order in which they are added in code. To improve consistency, try to add relevant exits in the following order: north, south, east, west, northeast, northwest, southeast, southwest, and then any other exits. No-one will shout at you if you don't do this, but it's the convention.

One thing you may have noticed from other parts of the game, especially in market squares, is that there are sometimes diagonal exits you can take that aren't listed – if you can go north and then east to get to a particular room, why can't you go northeast?

There's no reason why we should exclude this, so let's first add in the diagonal exits in the market rooms:

market_northeast

```
add_exit ("southwest", PATH + "market_southwest", "path");
```

market_northwest

```
add_exit ("southeast", PATH + "market_southeast", "path");
```

market_southeast

```
add_exit ("northwest", PATH + "market_northwest", "path");
```

market_southwest

```
add_exit ("northeast", PATH + "market_northeast", "path");
```

And next, we will set them to be hidden. Why do we do this? Mainly it's to avoid cluttering up the obvious exits list - whether you want to make your exits hidden is entirely up to yourself, but there's no reason why you shouldn't know how to do it. We make use of a new piece of code called modify_exit for this – modify_exit is tremendously powerful, and we'll return to it several times in the future.

In each of the market rooms, we'll modify the exit of the diagonal so that it is set as 'non-obvious'. It's still there, just not listed. For example, in market_northeast we'd just the following line of code:

```
modify_exit ("southwest", ({"obvious", 0}));
```

Do the equivalent of this in all four market rooms and update – you'll find the exits disappear from the list, but you're still able to take them just as before.

Chain, chain, chain...

Another thing you'll have noticed as you wander around the game is that some parts of various cities give you informs as to what's happening in another part... for example, you'll see something like

```
Drakkos moves southeast onto the centre of Sator Square.
```

It would be cool if our market rooms did that too – and they're going to by making use of a thing called the linker. First though, we need to make sure their shorts are set properly, because that's what's used to build the message. In each of the market rooms, change the shorts as follows:

market_northeast

```
set_short ("northeast corner of the marketplace");
```

market_northwest

```
set_short ("northwest corner of the marketplace");
```

market_southeast

```
set_short ("southeast corner of the marketplace");
```

market_southwest

```
set_short ("southwest corner of the marketplace");
```

For each of these, you should also set the determinate as 'the' instead of 'a'. In that way, people will see, for example, 'the northeast corner of the marketplace' when people move about. If it's set to 'a', then they'll see 'a northeast corner of the marketplace', which doesn't look right, not right at all!

Setting the shorts properly ensures that the messages will be properly formed, but still need to tell the MUD we want it to happen. We do this using set_linker. We give the MUD each room we want to link together except for the room in which we're defining the code.

For market_northeast then, the code would be as follows:

```
set_linker ( ({
 PATH + "market_northwest",
 PATH +"market_southwest",
 PATH + "market_southeast",
}) );
```

For market_northwest:

```
set_linker ( ({
  PATH + "market_northeast",
  PATH + "market_southwest",
  PATH + "market_southeast",
}) );
```

market_southeast:

```
set_linker ( ({
 PATH + "market_northwest",
 PATH + "market_northeast",
 PATH + "market_southwest",
}) );
```

And market_southwest:

```
set_linker ( ({
 PATH + "market_northwest",
 PATH + "market_northeast",
 PATH + "market_southeast",
}) );
```

You'll need to log on a test character to make sure that you've got this all setup properly, you can't test it from your own perspective. Create a test character, bring them into the world, and then set them as a test character using the testchar command:

```
testchar <testchar_name> on
```

Trans your testchar to your village and make them dance around a bit for you. What you should see are messages along the lines of the following:

```
Draktest moves south into the southwest corner of the marketplace.
```

Provided that message looks right from every part of the marketplace, you've got it all configured properly. Well done!

Conclusion

Having a skeleton of an area is a great way to give you a perspective of how it all fits together, as well as a solid idea of how much work you have to do. There's nothing wrong with the 'write one room and link it in' approach, but you get a much better idea of the bigger picture by architecting it all together and just seeing how it feels. That way, if you think that the layout needs to change, you can do it before you've hooked too much of it together. It's just a nice way to get some perspective.

My personal preference is to do this and then watch as the area starts to evolve in line with my development. It's very nice to be able to see an area taking shape before you. When we did the city of Genua, the outer circle of the city was all created and linked together before anything had really been done, and it was great to watch it slowly constructed around me as various creators went about their business. You should however find an approach that works best for you. Your mileage may vary, as they say.

Building The Perfect Beast

Introduction

It's awfully lonely in our little village, isn't it? I think it is, anyway – and surely, like I, you crave some kind of company on your quest to become a fully capable Discworld creator. In this chapter we're going to build the first inhabitant of our area, and provide him with equipment, responses, and chats. He's going to be our little living doll, for us to taunt and make dance for our sport.

In the process, we'll have to look at some new syntax and introduce a new concept of LPC programming, that of the variable. We've come quite far without actually talking abut variables, but you've been using them all along – yes, that's right! That's the SHOCKING TWIST of this chapter - variables are the Kaiser Soze of LPC programming!

A First NPC

Let's start off by creating a new subdir in our learnville directory – this one will be called chars:

```
mkdir /w/your_name_here/learnville/chars/
```

It is in this directory that we will include the code that creates any NPCs in our area. Keeping a clean divide between rooms, NPCs and items is an important part in ensuring it's easy to integrate your code into a larger domain plan. This is going to cause some complications for our path.h file, which we'll talk about a bit later – there is no problem so great that we cannot solve together, though!

Anyway, creating an NPC follows a very similar process to creating a room – what changes is the inherit we use, and the specific code that goes into the setup of the object. Consider the following for a basic template – save it into chars with the filename captain_beefy.c.

```
inherit "/obj/monster";

void setup() {
  set_name ("beefy");
  set_short ("Captain Beefy");
  add_adjective ("captain");
  add_alias ("captain");
  add_property ("determinate", "");
```

```
  set_gender (1);
  set_long("This is Captain Beefy, a former officer in Duchess Saturday's "
   "Musketeers.  He retired from there after being stabbed in the face by "
   "a marauding player.  He now lives in Learnville, hoping to the Gods "
   "that he will die in his sleep before he is murdered for his
   shoes.\n");
  basic_setup ("human", "warrior", 150);
}
```

Let's go through that line by line, as we did for our first room:

```
// If we want to create a basic NPC, this is the file we inherit.
inherit "/obj/monster";

void setup() {

  // The name of the NPC is how it is identified by the MUD's matching
  // system.  This should be one single word - it should usually too be
  // the last word in the short, for simplicity's sake.
  set_name ("beefy");

  // This is what players see when they encounter the NPC.
  set_short ("Captain Beefy");

  // This is what gets prepended to the short of the NPC.  Since Captain
  // Beefy is a unique person, he gets his determinate set to empty.  If
  // he was one of a number of clones (for example, a beggar), then
  // the determinate could be set to 'a', or even 'the'.
  add_property ("determinate", "");
  // Captain Beefy is a unique NPC - there should only ever be one of him.
  // This code doesn't ensure that, but it does mean when he's killed he'll
  // trigger a death inform.
  add_property ("unique", 1);
  // The name is used to match an NPC, but we also need to set valid
  // adjectives.  If we don't include this, our NPC can be referred to as
  // 'beefy', but not 'captain beefy'.  We want both to be valid.
  add_adjective ("captain");
  // We also want people to be able to refer to him just as 'captain', so
  // we add an alias for him.
  add_alias ("captain");
  // He needs a gender, because he's a he.  Setting the gender to 1 makes
  // him male.  2 makes him female.  Everything else makes him an 'it'.
  set_gender (1);
  // The long is what players see when they 'look' at him.
  set_long ("This is Captain Beefy, a former officer in Duchess Saturday's "
    "Musketeers.  He retired from there after being stabbed in the face by "
    "a marauding player.  He now lives in Learnville, hoping to the Gods "
    "that he will die in his sleep before he is murdered for his shoes.\n");
  // You need this in the code, or try as you might he won't clone when you
  // want him to.  The first piece of information we provide is his race.
  // The second is his guild.  The third is his guild level.
  basic_setup ("human", "warrior", 150);
}
```

So, update your code, and clone him – he'll appear in the same room as you if all has gone to plan:

```
The land is lit up by the eerie light of the waxing crescent moon.   This is
a skeleton room.
It is a freezing cold backspindlewinter's night with a steady wind, thick
black clouds and heavy snow.
There are two obvious exits: south and west.
Captain Beefy is standing here.
```

We can interact with him in the same way we can with any object in the game, but there's not much point. He doesn't do anything interesting at all. He just stands there looking gormless. But he loads! Chant it like a mantra, 'But He Loads!'. That's always the first promising step you take. After that, the rest is inevitable.

Breathing Life Into The Lifeless

First of all, let's make him emit some chats. Much like with our rooms, we can make our NPCs a little more interesting by adding chats. Moreover, we can make these chats different depending on whether the NPC is being attacked or not. The mechanisms by which we do this are identical in terms of syntax and meaning, but they're quite different from how it is handled in rooms. The code we need is called load_chat for normal chats, and load_a_chat for attack chat.

Let's add some chats into our NPC, and talk about what the code means:

```
load_chat(20, ({
  2, "' Please don't hurt me.",
  1, "@cower",
  1, "' Here come the drums!",
  1, ": stares into space.",
}));
```

This sets up the random chats for the NPC. The first number dictates how often a random chat is made. Every two seconds, (a period of time known as a **heartbeat**) the mud rolls a one-thousand sided dice (metaphorically), and if the result is lower than the number set here, the NPC will make a chat. It's not exactly fine-grained control, but control it is.

Each of the chats has two parts - a **weighting**, and a command string. The weighting is the relative chance a chat will be selected. The string is the command that will be sent for the NPC to perform. If you want the NPC to say something, then start the command with an apostrophe. If you want the NPC to emote, then start the command with a semi-colon. If you want the NPC to perform a soul, start the command with an at symbol.

To understand the way the weightings work, think of it as a roulette wheel – add up the weightings of all the chats, and it'll come to 5. When the MUD determines the NPC should make a chat, there is a 2/5 chance it'll be the first chat, and a 1/5 chance it'll be each of the others. As usual, there is more that you can do with load_chat than we've covered here – we'll get to some advanced stuff later in the tutorials.

One proviso here is that if you want your NPC to actually say things, it's going to need a language and a nationality. We can provide that by using the setup_nationality method providing in an appropriate combination of nationality and region. Your domain leader will be able to tell you which of these is appropriate for your domain, but for demonstration purposes we'll make Captain Beefy a Genuan. Put it after basic_setup in your code:

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

To add the attack chats, it works the same way – we just use load_a_chat instead:

```
load_a_chat(40, ({
  1, "' Oh please, not again!",
  1, "' I don't get medical insurance!",
  1, "@weep",
}));
```

When you make changes to Beefy, you'll need to dest him before you update and reclone. Once you've added the chats, go on – take a swing at him! You may find he doesn't swing back – if you're invisible, that'll be why. But you should find he pleads pitifully before your Awesome Might:

```
You punch at Captain Beefy but he dodges out of the way.
Captain Beefy exclaims: Oh please, not again!
```

Poor fellow. He doesn't know what Fresh Horrors we have in store for him to come.

So, that's fine for making him a little more interesting – however, we also want to be able to make him reply when we say things to him. Let's add some of that now using add_respond_to_with. The syntax for this is a little complex, so bear with me:

```
add_respond_to_with(
  ({
    "@say",
    ({"hello", "hi", "hiya"}),
  }),
   "' Er, hello.  Please don't kill me.");
```

This sets up a response to a message that originates with the command 'say'. It will match on any string containing either the words 'hello', 'hi', or 'hiya', provided it originates from a say command. Beefy's response to this will be to say "Er, hello. Please don't kill me."

Let's add in a second response, since he's already opened us up to the possibility of killing him:

```
add_respond_to_with(
 ({
 "@say",
 ({"kill", "murder"}),
 }),
 "' No, please no.  I beg you!");
```

Pitiful, isn't it? However, it's also not quite what we want – because if we say 'I am not going to murder you', he'll still beg for his life. Really we only want him to make this response if we suggest we are going to murder him. We can do this by adding in a second set of keywords. The MUD will attempt to pattern match based on the order we give the sets of keywords: This kind of pattern matching is not easy – the more complex you make the trigger conditions, the less likely people will be able to hit on them properly.

The MUD will also only execute one match, so the order in which you add the triggers will influence the way the NPC responds – if you have a more specific response you want to catch before a more general one, it should go first in your code:

```
add_respond_to_with(
  ({
    "@say",
    ({"won't", "not", "can't"}),
    ({"kill", "murder"}),
  }),
   "' Thank you, kind sir!");

add_respond_to_with(
  ({
    "@say",
    ({"am", "will", "going"}),
    ({"kill", "murder"}),
  }),
  "' No, please no.  I beg you!");
```

You'll get one responses here to 'I am not going to kill you', and a different one to 'I am going to kill you'. Try them the other way around, and you'll see a quite different story unfold!

These are, of course, terrible responses because they don't really catch meaning – if I say 'I'm not going to not kill you', he'll still thank me for my mercy. Keeping your responses simple will help you manage that kind of thing - convincing conversations are not easy to do, and since we work on keyword triggering, there is no way for us to derive semantic meaning. Make your responses as complex as they need to be, but no more complex than that.

We can add a response that triggers on a variety of different triggers – for example, to get him to respond to some souls:

```
add_respond_to_with(
  ({
    ({"@comfort", "@soothe", "@calm"}),
    ({"you"}),
  }),
  ": takes a deep breath.");
```

Notice that the trigger text for this is the word 'you' – that's because that's what the NPC sees from its perspective.

We can also make him give random responses by slightly changing the format of the last part of the add_respond_to_with. For example, let's change our last add_respond_to_with a little:

```
add_respond_to_with(
  ({
    ({"@comfort", "@soothe", "@calm"}),
    ({"you"}),
  }),
  ({
    ": takes a deep breath.",
    "' Yes, you're right... I shouldn't let things get to me."
  })
);
```

He'll now respond randomly with one of the two chats we've given him.

Cover Yourself Up, You'll Catch A Cold

Captain Beefy will now chat with us a bit, but he's still horribly naked. That's embarrassing for everyone concerned, so let's dress him up a bit, like the little doll he is. This is where we counter our first real bit of LPC – the variable, and the first handler we'll deal with.

The handler is question is called the armoury, and it's what we use to get hold of game objects. Your request command makes use of the armoury, and you can use it to provide a list of the things available to you. Before we use the armoury, we need to add something to the top of our code – another #include directive. This time, we need to tell our code where it can find the armoury:

```
#include <armoury.h>
```

Notice that the name of the .h file is enclosed in angle brackets rather than quotation marks. If you surround the name in angle brackets, the driver looks for the file in /include/ first, and then in its current working directory. If you surround the name in quotation marks, it looks in the working directory first and *then* in the /include/ directory.

In this .h file is a define for ARMOURY – this points to the object in the mudlib that manages keeping track of items and making them available to other objects. That's what a handler is – an object that has responsibility for managing some aspect of the game so that it's easier for other objects.

As mentioned elsewhere in this material, everything in Discworld is an object. Captain Beefy is an object, and so is every item in the game. When we want to refer to one of these objects in code, we need to get a reference to it.

First of all, we need to define something to hold that reference – we use a variable for that. At the top of your setup for Captain Beefy, add the following line of code:

```
object trousers;
```

This is a variable declaration. This requests a little portion of the computer's memory from the driver, and that portion of memory is exactly big enough to hold a MUD object regardless of what kind of object it actually is.

It doesn't actually have anything in it yet though – we first have to set its contents through what is known as an assignment. All variables are defined at the top of whatever function they happen to be in (more on this later) – so they go before any other code in your setup. They are usually assigned later in the code – so while the variable declaration occurs at the top of the code, we'll put something into that variable after we've done all the rest of the setup for the NPC:

```
#include <armoury.h>

inherit "/obj/monster";

void setup() {
  object trousers;
```

```
  set_name ("beefy");
  set_short ("Captain Beefy");

  // REST OF THE CODE HERE

  add_respond_to_with(
    ({
      ({"@comfort", "@soothe", "@calm"}),
      ({"you"}),
    }),
    ({
       ": takes a deep breath.",
       "' Yes, you're right... I shouldn't let things get to me."
    })
  );

  trousers = ARMOURY->request_item ("pirate trousers", 100);
  trousers->move (this_object());

  init_equip();
}
```

Let's look at what our new code does:

```
trousers = ARMOURY->request_item ("pirate trousers", 100);
trousers->move (this_object());

init_equip();
```

The first line here sends a request to the armoury for an item with the name 'pirate trousers'. You can find out all the trousers that the armoury has available using the command:

```
request list clothes trousers
```

These trousers were chosen at random – feel free to pick whatever pair appeals to you. At this point, a clone is made of these trousers, but they don't exist anywhere except in the computer's memory. The second line takes those trousers and moves them into the inventory of our NPC.

The third line makes the NPC wear and hold all of the equipment it is currently carrying – it makes him dress himself, essentially.

Let's do the same thing with a shirt. Create a variable at the top:

```
object shirt;
```

And then after we've requested our trousers, we request a shirt:

```
shirt= ARMOURY->request_item ("white ruffled shirt", 100);
shirt->move (this_object());
```

This should come before the init_equip(), which we only need once.

You can clone weapons, food, scabbards, jewellery – practically anything you may like – into your NPC in this way. Have a play about with the request command to see what's available to you.

Request Item

The above syntax is a little more complicated than it needs to be - we also have access to a piece of code called request_item which does all of this work for us. It still uses the armoury, it just does more of the work for us. It creates the variable, requests the item from the armoury, and then moves it into our NPC. We don't need the #include, or the variables, we just need to use request_item directly:

```
request_item ("pirate trousers", 100);
request_item ("white ruffled shirt", 100);

init_equip();
```

However, you'll see an awful lot of NPCs doing it the 'longhand' way, so you should understand how both mechanisms work. You will often find situations on Discworld where there are easier ways to achieve what a piece of code is doing, and they are usually down to one of the following reasons:

- The creator in question didn't know about the easier way.
- The easier way was added after the code was written

Because there are so many creators who have contributed so much code over so many years, there is a veritable archaeological record in our mudlib and domain code. You need to know how both ways work because you're just as likely to encounter a complicated way of doing things as you are an easier way.

Chatting Away

There's one thing left for us to talk about in this chapter, and it's the special codes that can be used within load_chat and load_a_chat to make our NPCs more involved in the world around them. We can build special strings that get interpreted by the MUD and turned into meaningful output based on the things around our NPC. It's easier to see what that means in practise than describe it, so let's add a load_chat to Captain Beefy:

```
load_chat(80, ({
  2, "' Please don't hurt me.",
  1, "@cower",
  1, "' Here come the drums!",
  1, ": stares into space.",
  1, "' Oh, hello there $lcname$.",
}));
```

See the last chat there? The weird looking code is interpreted by the MUD to take the form of the name of a living object in the NPC's inventory. The first letter (l) defines what object will be the target of the chat, and the string that follows (cname) defines what is used to build the rest of the string. When the chat is selected, a random object is picked from the specified set, and then the requested query method is called on it and substituted for the special code we provide. The letters we have available for choosing the set of objects is as follows:

Letter	Object
m	The NPC itself.
l	A random living object in the NPC's environment.
a	A random attacker in the NPC's environment.
o	A random non living object in the NPC's environment.
i	A random item in the NPC's inventory.

Following the initial letter comes the type of information being requested. This gets called on the random object that is selected when the chat triggers. Some of these are more useful than others, but you may find cause to use even the more specialised ones on occasion:

Code	Request
name	The file name of the NPC - used for targeting souls
cname	The capitalised name of the object
gender	The gender string of the object
poss	The possessive string
pronoun	The pronoun of the selected object
ashort	The a_short() of the object
possshort	The poss_short of the object
theshort	The the_short of the object
oneshort	The one_short of the object

As an example, we could get the short of an object in the NPC's inventory with $itheshort$. We could get the name of a random attacker in load_a_chat with $acname$. Unfortunately, we get no fine-grained control over the object selected - if we choose the item set of objects, we can't further specialise it, so a chat like the following will not be appropriate:

```
"' I am going to stab you in the eyes with $ipossshort$!"
```

It'll parse properly, but he'll end up saying things like 'I am going to step you in the eyes with my floppy clown shows!' which, while surreal, is not really sensible. Despite the limitations, combining these codes will allow for you to make your NPC chats more dynamic and responsive to the context in which it finds itself.

Conclusion

We've come quite far in this chapter, having created an interactive NPC who is dressed in a fashion that does not offend our Victorian sense of decency. However, we need to manually clone Captain Beefy into our village each time we want him there. In the next chapter we'll look at how we can get that to happen automatically without our intervention.

At the moment we're making use of only those items that the armoury can provide, but we'll also look at ways in which we can write our own objects and make them available to our NPCs and our rooms.

Hooking Up

Introduction

So, we now have a set of rooms, and we have an NPC. They're simple, but they work. It's not appropriate though that you have to clone the NPC directly into your room – it should happen automatically, and we're going to talk about how that works in this chapter.

We're also going to resolve the path.h problem that we introduced in the last chapter by looking at relative and absolute directory referencing. So buckle up, time to take LPC out for another spin!

The Path.h Problem

Look at where we've got our path.h file stored – it's in our rooms directory. Although we haven't needed to refer to it in the NPC we created, we should still be able to get access to it without too much complication... the idea of a header file is that it's shared between all relevant objects, after all.

We have a couple of possibilities. One is to copy the path.h file into each directory that we are likely to need it. This is a bad solution because it reintroduces the problem it was designed to fix – we need to change the defines in multiple places if we want to shuffle things around. That's not ideal – we want to be able to change things in one place and have it reflected in all appropriate locations.

Our second possibility is to make everything use the same path.h file – that's a better solution, but it's going to need us to change all the references to the path.h in all our code. We'll need to put it in a central location, and then have all of the files #include it from there. That's not a bad solution - there are plenty of .h files in /include/ that work this way, but you need special file access to put files in there, and comparatively few creators have that access. We'd need to store it then in a set directory in our /w/ drive. We can do better than that though – after all, if we move our code from /w/ to /d/, we're going to have to remember to move the .h file along with it, and then change all the references to the path.h file to reflect its new location Ideally it should just be a case of copying a directory across and having done with it.

How about this though – we keep a path.h in each subdirectory, but we have that path.h in itself include a path.h in a higher level directory? That way, provided the basic structure of the directory remains intact, we have a chain of path files that define all the values we need.

That may sound confusing, but let's see it in practice – it should become a bit clearer with an example.

Let's start with a new path.h file in our base learnville directory:

```
#define     PATH          "/w/your_name_here/learnville/"
#define     ROOMS         PATH + "rooms/"
#define     CHARS         PATH + "chars/"
```

In each of your subdirs, you add a further path.h file that includes the path.h from the higher level directory. We can do that using the .. symbol to to represent the higher level directory in the #include. The double period symbol has a special meaning in a #include - it means 'go to the directory one up from the current directory':

```
#include "../path.h"
```

Now, we are going to have to change our room code a bit because we're making a distinction between ROOMS and CHARS. Luckily we don't need to do that by hand, we can use the sar command to do a search and replace. I shall warn you in advance though, be VERY CAREFUL when using this command. One creator, who shall remain nameless (Terano) once mistakenly changed every instance of the letter a in all the priest rituals to the letter e. While tremendously funny (to everyone else), it was hugely problematic for him to fix.

Anyway, the sar command needs three pieces of information – the string of text you want to replace (surrounded in exclamation marks), the string of text you want to replace it with (again, surrounded in exclamation marks), and the files you want the text replaced in. Let's run that puppy over our code. First, you change your your current directory to the rooms subdirectory, and then:

```
sar !PATH! !ROOMS! *.c
```

Upon uttering this mystical incantation, you'll find all of your rooms now reflect the New Regime. Update them all (you can do this using update *.c), and you'll find everything is Hunky Dory.

If it's not, remember what we discussed about path.h files in a previous chapter - if they don't work properly, make sure there's a carriage return at the last line. Sometimes LPC chokes on a file if that's not the case.

With regards to sar, please remember – use this command with caution. It's incredibly easy to do some really quite impressive damage to your hard work with only a few misplaced keystrokes. You Have Been Warned!

Sorted!

Right, now we've gotten that out of the way, let's look at how we can make Captain Beefy appear in our rooms. We're going to pick a room for him (we'll choose market_northwest) and talk about how it works. First of all, we need to talk about a new kind of programming concept - the function. A function is essentially a little self-contained parcel of code that gets triggered upon request - sometimes on our request, sometimes on the request of another object in the MUD. We don't need to worry too much about it just now, we just need to know that's what we're about to do. Functions have many different names - they most common one you'll also see in this material is the word 'method'. It's just another word for the same thing.

There are certain functions that the MUD executes on all Discworld objects at predetermined intervals. One of these we're familiar with – the code that belongs to setup gets executed when we update our objects. There's another function that gets called at regular intervals, and that's the reset function. The reset function is called on rooms when a room is loaded (and it's called just after setup), and also at regular intervals (of around thirty minutes or so) on all currently loaded rooms. Just the place to deal with loading an NPC!

So, we're going to add a new function to market_northwest, like so:

```
void reset() {
  ::reset();
}
```

This function exists **outside** of any code you've already got. So within your room, it will look like this:

```
void setup() {
  ...
}

void reset()
{
  ::reset();
}
```

Yikes! What does that code inside it mean? Don't worry too much about that right now – in brief, in the object you've inherited at the top of your object, there's already a reset function defined. What we're saying with that line of code is 'Oh, remember and execute all the code that's in the other reset method too'.

Now, we've already seen how to create a variable to hold an object. NPCs are objects too, and so inside our reset method we want a variable that can hold Captain Beefy:

```
object beefy;
```

Remember, this goes at the top of our function, before the ::reset().

After the ::reset(), we start building our code. The process for loading an NPC into a room is as follows:

```
Load the NPC and assign it to a variable.  If there is something in that
variable, and that variable's location is not this room, then move the NPC
into the room
```

We don't load NPCs in the same way we get items from the armoury – instead, we use a piece of code called load_object, passing it the filename of the object we want to load.

```
beefy = load_object (CHARS + "captain_beefy");
```

What comes out of that code is the reference to the loaded version of Captain Beefy, and we store that reference in our beefy variable. We tie these together like so:

```
void reset() {
  object beefy;
  ::reset();

  beefy = load_object (CHARS + "captain_beefy");
}
```

After_reset also exists separately from your existing functions, like so:

```
void setup() {
  ...
}

void reset() {
  ...
}

void after_reset() {
  ...
}
```

This doesn't actually move Beefy into our room – in order to do that, we need to explore a new element of programming syntax – the if statement.

If At First You Don't Succeed

The if statement is the first programming structure we're going to learn how to use. It allows you to set a course of action that is contingent on some preset factor. The basic structure is as follows:

```
if (some_condition) {
  some_code;
}
```

The condition is the important part of this – it's what determines whether the code between the braces is going to be executed. A condition in LPC is defined as any comparison between two values in which the comparison may be true or false. If the comparison is true, then the code between the braces is executed. If the comparison is false, LPC skips over the code in the braces and instead continues with the next line of code after the if statement.

The type of comparison depends on which of the comparison operators are used... these go between the two values to be compared, and determine the kind of comparison to be used:

Comparison operator	Meaning
==	Equivalence – does the left hand side equal the right hand side?
<	Is the left hand side less than the right hand side?
>	Is the left hand side greater than the right hand side?
<=	Is the left hand side less than or equal to the right hand side?
>=	Is the left hand side greater than or equal to the right hand side?
!=	Does the left hand side not equal the right hand side?

Let's look at a simple example of this in practise using whole numbers (the int variable type):

```
int num1;
int num2;

num1 = 10;
num2 = 20;
```

```
if (num1 < num2) {
  tell_creator ("your_name_here", "num1 is less than num2!\n");
}
tell_creator ("your_name_here", "I'm out of the if!\n");
```

So, if the value contained in the variable num1 is less than the value contained in the variable num2, then we see the message sent to our screen. If it isn't, then we don't. In either case, we'll see the "I'm out of the if!" message. So, with the values we've given num1 and num2, our output is:

```
num1 is less than num2
I'm out of the if!
```

If we change the two variables around a bit:

```
num1 = 20;
num2 = 10;
```

Then all we see is:

```
I'm out of the if!
```

An if statement by itself allows for you to set a course of action that may or may not occur. We can also combine it with an else to give two exclusive courses of action:

```
if (num1 < num2) {
  tell_creator ("your_name", "num1 is less than num2!\n");
}
else {
  tell_creator ("your_name", "num1 is greater than or equal to num2!\n");
}

tell_creator ("your_name", "I'm out of the if!\n");
```

So now, if the condition is true, we'll see:

```
num1 is less than num2
I'm out of the if!
```

And if it's not, we'll see:

```
num1 is greater than or equal to num2
I'm out of the if!
```

We can also provide a selection of exclusive courses of action by making use of an else-if between our original if (the one that starts the structure) and the concluding else (if we want one – else is always optional):

```
if (num1 < num2) {
  tell_creator ("your_name", "num1 is less than num2!\n");
}
else if (num1 == num2){
  tell_creator ("your_name", "num1 is equal to num2!\n");
}
else {
  tell_creator ("your_name", "num1 is greater than num2\n");
}
```

We can add as many else-ifs as we like into the structure until we get the behaviour we're looking for.

So, that's what an if statement looks like. Let's tie that into our reset function above. Any variable that does not have anything in will have a null value (we can use 0 to represent this in an if statement). So if we want to know if our beefy variable contains an actual Captain Beefy:

```
if (ob != 0) {
  some_code;
}
```

We can even write this in a simpler fashion – LPC lets us check to see if a variable has a value set by simply including it as its own condition in an if statement:

```
if (ob) {
  some_code;
}
```

So, that puts us firmly in the position of having met the first of our requirements to move our Captain into the room. Now, let's look at the second requirement.

We can do that too by putting an if statement inside our if statement – this is known as nesting. To tell whether or not Captain Beefy is in the same room as the code we're working, we use the following check:

```
if (environment (beefy) != this_object()) {
}
```

If both of those things are true, then we can move our Dear Captain into the room:

```
if (beefy) {
  if (environment (beefy) != this_object()) {
    code_to_move_beefy;
  }
}
```

This is not ideal though – in general, having nested structures leads to clunky, inelegant code. There are sometimes very good reasons for code to be nested, but if you don't have to do it, you shouldn't. In this case, we wouldn't have to do it if we could get one if statement to check for both things. Luckily, that's something we can indeed do!

Compound Interest

We're not restricted to having a single condition in an if statement – we can link two or more together into what's called a compound conditional. To do that, we need to decide the nature of the link.

Things become more complicated the more conditions that are part of a compound – we can have as many as we like, but let's start out as simply as we can. Because we only want to execute the code in our if statement if both conditionals are true, we use the **and** compound operator. In LPC, this is represented by a double ampersand: &&. If we wanted the code to be executed if one condition or the other were true, we'd use the **or** operator, which is a double bar: ||.

We can join our two if statements together into one Beautiful Whole using our and operator:

```
if (beefy && environment (beefy) != this_object()) {
  code_to_move_beefy;
}
```

That's much neater all around.

It's sometimes confusing to new coders which of the compound operators they want for a particular situations. There is a concise representation of what each of these conditions means – it's called a truth table. The truth table for and is as follows:

First Condition	Second Condition	Overall Condition
false	false	false
true	false	false
false	true	false
true	true	true

This means that if both conditions in the compound evaluate to true, only then is the overall condition that governs the if statement evaluated to true. In all other cases, it is evaluated to false.

For or, the truth table looks like this:

First Condition	Second Condition	Overall Condition
false	false	false
true	false	true
false	true	true
true	true	true

More complex conditionals can be built by linking together conditional operators. That's a discussion for a later chapter though.

Now that we have our if statement, we can look at the code we actually need to move Captain Beefy into our room.

Moving

There's a piece of code defined in all items, living or otherwise, and that code is called move – we use that to move NPCs from one place to another. For Captain Beefy, it looks like this:

```
beefy->move (this_object(), "$N appear$s with a pop.",  "$N disappear$s with a pop.");
```

The first part, this_object() refers to where we want the NPC to move. In this case, it's the room in which we're currently working. The second is the message people will see when the NPC moves into the room. I know it doesn't look like a normal message, but we'll come back to that. The third part is the message people who are currently with the NPC will see when it is moved.

The move messages work using a special kind of notation that is interpreted by the MUD to form the output properly depending on the perspective of the observer. The perspective doesn't mean much to an NPC, but it makes all of the difference when it comes to moving players around. To an outside observer, $N gets displayed as the short of the object being moved. To the object being moved, it gets displayed as 'You'.

The word appear$s works in a similar way. An outside observer will see the word appear with the s appended to the end (appears). The object being moved will see only 'appear'.

So, if Captain Beefy were a real person, he'd see:

```
You appear with a pop.
```

Everyone else sees:

```
Captain Beefy appears with a pop.
```

The same system is used for the message of him disappearing – it just makes the whole thing look much nicer.

Putting that all together in our reset function gives us the following:

```
void reset() {
  object beefy;
  ::reset();

  beefy = load_object (CHARS + "captain_beefy");

  if (beefy && environment (beefy) != this_object()) {
    beefy->move (this_object(), "$N appear$s with a pop.",
      "$N disappear$s with a pop.");
  }
}
```

Update the room, and you'll see Captain Beefy is there with you! That's nice, but we have one final cosmetic touch to include.

One Final Touch

When the room is loaded, there is no entry message for Captain Beefy. You can prove it works by desting beefy and then using the call command to force a reset:

```
call reset() here
```

You'll see the message we set in our move:

```
Captain Beefy appears with a pop.
```

We want that to happen when the room is updated as well, but it doesn't. In order to make it happen, we have to delay the creation of the NPC a little bit.

There is a special function defined by the driver called 'call_out' – it lets you provide the name of a function, and a delay. After the number of seconds indicated by the delay, the named function is called. The convention for this behaviour in terms of reset is to have the actual functionality moved into a function called after_reset. After that's done, your code will look like this:

```
void reset() {
  ::reset();
  call_out ("after_reset", 3);
}

void after_reset() {
  object beefy;
  beefy = load_object (CHARS + "captain_beefy");
  if (beefy && environment (beefy) != this_object()) {
    beefy->move (this_object(), "$N appear$s with a pop.",
      "$N disappear$s with a pop.");
  }
}
```

It should be noted at this point that we are not yet talking about why certain parts of the code need to be in certain places. We'll get to that, don't fret.

Conclusion

We've now hooked up our NPC and our rooms – and in the process incorporated a header file that spans multiple directories. Not only is our area starting to shape up in terms of the contents and the features, we're doing it in such a way as to guarantee the maintainability of the code. That's incredibly important, although I appreciate it may appear underwhelming for now.

We're still not talking much about code, although you've now been introduced to the first of your programming structures – the function, and the if statement. You've reached a point where you now have the capability to make objects react intelligently to the circumstances in which they find themselves – that's tremendously powerful! Onwards and upwards!

Back To The Beginning

Introduction

In this chapter, we're going to take a further look at the code we can make use of in our rooms. Hardly any of our rooms have any descriptions, and we still need to discuss some of the cooler things that can be done with add_item, as well as the way in which we can provide more realistic descriptions by incorporating the changes between night and day. This requires us to write rooms with two sets of descriptions, but the effect is really very appealing.

We're also going to make Captain Beefy wander around this fine village of ours, and add a special skill based search to one of our rooms. It's all very exciting! Touch your nose!

Captain Beefy's Return

First, we're going to make Captain Beefy wander around our village – after all, it gets so lonely when we're left by ourselves. NPCs wander according to a series of move zones that are defined. They are defined firstly in themselves (to determine what zones an NPC may roam within) and secondly in the rooms (to define to which zone a room belongs). We're going to define all of Learnville as a single zone, so add this to each of your rooms, somewhere in the setup function. It doesn't especially matter where.

```
add_zone ("learnville");
```

We can add multiple zones to a room, allowing for NPCs to have shared but distinct wandering areas.

Once you've added that zone to each room, we need to add the correct zones to Captain Beefy. In his setup, add the following:

```
add_move_zone ("learnville");
set_move_after (60, 30);
```

We use add_move_zone to setup which zones within which our NPC is permitted to roam. We use set_move_after to set the speed Beefy will wander – the first number sets the fixed delay, and the second sets the maximum number of random seconds that is added to that delay. With that code, Beefy will wander every 60 to 89 seconds.

That's enough to set Beefy wandering around our village. It's quite a hassle to manually add a zone to every room – there are ways and means by which the Industrious Creator can avoid this hassle, but they're a bit too advanced for us at the moment. We shall thus simply live with the inconvenience. LPC For Dummies 2 will open up new worlds of shared functionality for us.

The Road Less Travelled

We're going to return to street_03 here – it's still set as a skeleton room and has no long description, or any items. We're going to use this blank canvas as the exploration point for some new functionality.

First of all, what we've done for the items and long in street_01 isn't, strictly speaking, correct. Oh, it works and it does what we said it would, but it doesn't capture the dynamism that we normally associate with Discworld MUD. On your travels, you have undoubtedly noticed how in many areas the room descriptions, add_items and even chats in a room are different when it's night to when it's day. All rooms on the MUD should include this basic level of responsiveness to the world.

Instead of using set_long, we use a pair of related methods – set_day_long and set_night_long. Functionally, they are identical to set_long except that they are time of day dependant. The MUD itself handles all the switching between the right version of the long, you just need to tell it what the long should be. Like so:

```
#include "path.h"

inherit "/std/room/outside";

void setup() {
  set_short ("simple village road");
  add_property ("determinate", "a ");
  set_day_long ("This path is lit by bright, beautiful daylight.  "
    "From the sun.  High above.  Because it's daytime, see?.\n");
  set_night_long ("It's dark here, because it's night-time.  As opposed "
    "to day time. Do you understand what I mean?");
  set_light (100);
  add_zone ("learnville");
  add_exit ("east", ROOMS + "market_southwest", "path");
  add_exit ("southwest", ROOMS + "street_02", "path");
}
```

Note that we don't use set_long at all – we only use that for rooms in which the description does not change at all from day to night – an underground passage, for example, would fit that bill.

Similarly, we can add day and night items to reflect the changing situations described in our longs:

```
add_day_item ("bright beautiful daylight", "It illuminates the world "
"around you!");
add_day_item ("sun", "It burns our eyes, precious!");
add_day_item ("daytime", "That's what it is.");
add_day_item ("nighttime", "Don't worry, it'll probably be daytime "
  "forever.  No need to fret.");
add_night_item ("nighttime", "It's night, alright.  You can tell by all "
  "the dark around.");
add_night_item ("dark", "You can't see the dark, because it's too dark.");
add_night_item ("daytime", "Those halcyon hours are gone for good, or at "
  "least until the sun comes up again.");
add_day_item ("sun", "There's no sun, because it's NIGHT.");
```

If we want things that are available for day and night, we just use a normal add_item - but only if they shouldn't change their appearance.

We can also enrich our rooms with alternating day and night chats:

```
room_day_chat ( ({ 120, 240, ({
  "The daytime is full of sunlight.",
  "You can see everything around you, because of the sun.",
  "The sun is shining in the sky.",
})}));

room_night_chat ( ({ 120, 240, ({
  "It is pitch black.  You are likely to be eaten by a grue.",
  "Was that a grue? It sounded like a grue.",
  "If that's not a grue you can hear, it might be Vashta Nerada.",
})}));
```

Obviously these are all terrible descriptions and violate all conventions of what you should put into long descriptions, items and chats – that doesn't matter for now because our focus is on function not form. Simply providing day and night descriptions goes a long way to increasing the sense of richness people experience in your areas, and you should definitely get into the habit of writing them. It adds a fair bit of extra work to room development, but the payoff is worth it. It's one of the reasons why our MUD looks so much slicker than many.

Sadly, we have to wait for the hours to tick by before the MUD swaps from night to day, so let us leave our descriptions there. You can use the **check** command to verify that they are present, but you'll need to wait until the time of day changes before you can come back and see them in their proper context. So let's move on to a different topic while we wait for the cruel, unyielding sun to set on our development.

Bigger, Better, Faster

Earlier in these documents, I mentioned that add_item was tremendously powerful. It is – it lets you do all sorts of things with your items that you may not have realized. In this section, we're going to look at some of the things that add_item lets us do. Note that we need to use a more complicated version of the add_item syntax to do all of these things – rather than just giving the name of the item and its description, we need to specifically state which parts of the add_item we're setting.

First of all, if an item is large and solid enough for people to sit, or stand, or kneel on – we should let them do that. We do that by adding a position tag to the item, giving the string that should be appended to the position, like so:

```
add_item ("jagged rock",
  ({
    "long", "This is a jagged rock.",
    "position", "on the jagged rock"
  }));
```

Add this to your street_03 room, and then update. You'll now find that you can 'sit on jagged rock' – but careful, that's bound to hurt. You can lie on it, stand on it, kneel on it... the usual suspects in terms of interaction choices. If you sit on it, everyone will see something like 'Drakkos is sitting on the jagged rock.". The text you set in the add_item is what defines how that message appears.

You can add interaction options to the items too – for example, if I wanted to make it possible to kick the stone, I add a kick tag to the code:

```
add_item ("jagged rock",
  ({
    "long", "This is a jagged rock.",
    "position", "on the jagged rock",
    "kick", "Ow!  That stung!\n"
  }));
```

Note that you need to end any specific verb response you define with a newline character. There's no limit to what verbs you may include – it's just a string of text that gets shown to the player when they attempt to use that verb on your item.

You can provide synonyms for verbs too:

```
add_item ("jagged rock",
  ({
    "long", "This is a jagged rock.",
    "position", "on the jagged rock",
    ({"kick", "punch"}), "Ow!  That stung!\n",
  }));
```

There is also a special tag called searchable that lets you set an add_item as responding to the search command. This is a little more complicated than just providing a string of text to respond with – you need to define a function in your room to handle the searching. We're going to do that next, but let's add the bit we need to the item first:

```
add_item ("jagged rock",
  ({
    "long", "This is a jagged rock.",
    "searchable", "#search_rock",
    "position", "on the jagged rock",
    ({"kick", "punch"}), "Ow!  That stung!\n",
  }));
```

The #search_rock section tells the MUD what function it is to call when someone attempts to search the rock. Much like with reset and after_reset, we need to provide the code to handle this, but the MUD ensures it gets executed at the right time. You can do the same thing with any verb – the # notation lets you define a function for each of these.

Probing Dark Depths

Let's start off with a very simple definition of the search_rock function – it won't do much at all, but it'll verify that what we have is correct so far:

```
int search_rock() {
  tell_object (this_player(), "There doesn't seem to be anything in the "
    "rock.\n");
  return 1;
}
```

The tell_object function sends a message to the object provided as its first piece of information – this_player() is a special piece of code that refers to whatever player (or NPC) was responsible for causing the code to trigger. We'll return to this_player() somewhat later – it's a bit more complicated than I have made out here.

The second part of tell_object is the text we want to send to the specified object.

Update the room, and search the rock. You should see the message you set echoed back to you. That gives us our starting point – we're going to do something a bit more adventurous though. Searching is no fun unless you have a chance of finding something interesting.

Here's what we're going to do – we're going to do a skill check on a player to see whether or not they find the secret hole in the rock. If they do, we will reward them with a Shiny Genuan Cent. If they don't pass the check, they don't get a thing. Also, we're going to make it so that this shiny coin can only be found once per reset period – that stops people continually searching the rock for infinite (albeit slowly accumulated) money.

It's a fairly complex task, one that requires us to investigate a few new bits of syntax and make use of a new MUD handler – the taskmaster.

The Taskmaster

The taskmaster is the thoroughly ingenious piece of code that is responsible for determining whether or not skill checks pass or fail, and providing skill awards where appropriate. In order to make use of it, we need to include the right header file at the top of our room:

```
#include <tasks.h>
```

The taskmaster has a number of methods that can be used to perform a skillcheck. We're going to investigate the simplest syntax, that which is used to make a check against some preset difficulty factor.

The code we need for this is perform_task, and is used like so:

```
int success;
success = TASKER->perform_task (this_player(), "other.perception", 100,
  TM_FIXED);
```

The first bit of information we provide is the object against which we check the skill (in this case, it's this_player()). The second is the skill we want to check against (other.perception). The third is the bonus at which the task has a chance of succeeding. The last is a special value that determines how likely a skill increase is... ignore that for now, we'll just put it as TM_FIXED (that's a define set in tasks.h).

The value that comes out of perform_task is a number that indicates the result of the check – at its most basic, it's either an AWARD (a tm increase), SUCCEED (the check passed), or FAIL (the check failed). We need to provide the appropriate behaviour to deal with the result. We can do it using the syntax we already know, but it's a bit clumsy:

```
int search_rock() {
  int success;
  int found;

  success = TASKER->perform_task (this_player(), "other.perception",
    100, TM_FIXED);

  if (success == AWARD) {
    tell_object (this_player(), "%^YELLOW%^%^BOLD%^You feel a little more "
      "perceptive.\n%^RESET%^");
    found = 1;
  }
  else if (success == SUCCEED) {
    found = 1;
  }
  else {
    found = 0;
  }

  if (found) {
    tell_object (this_player(), "You have found a shiny Genuan cent!\n");
    this_player()->adjust_money (1, "Genuan cent");
  }
  else {
    tell_object (this_player(), "You don't find anything in the rock.\n");
  }

  return 1;
}
```

Note that if we have an award, we provide the TM message and we also have to colour it – that weird looking collection of symbols (known as Pinkfish Colour Coding) is how we add colour to a message. In general, you don't do this. Colour is problematic for clean MUD design, so unless you have a reason, you shouldn't. Taskmaster awards are one of the few areas that are an exception to this general rule.

Since we need to handle it for a taskmaster message, you surround the text you want to change the colour of in this special code, and end it with a %^RESET%^. If you wanted to display something in red, you'd do:

```
%^RED%^Bing!%^RESET%^
```

This is something you should recognize rather than use yourself – the sole exception in day to day creating life is in a TM message.

Switching Things Around

The structure we have in place here is rather clunky – luckily LPC provides us with a more elegant alternative – the switch statement. A switch is essentially a compact representation of a complex if, else-if, else structure. First, we choose a variable to switch on – in this case, it's *success*. We then define a case for each of the possible alternate values the switch variable may have. The code that follows the case will be the code that is executed if the switch variable has the specified value.

```
success = TASKER->perform_task (this_player(), "other.perception", 100,
  TM_FIXED);

switch (success) {
  case AWARD:
  tell_object (this_player(), "%^YELLOW%^%^BOLD%^You feel a little more "
    "perceptive.\n%^RESET%^");
  case SUCCEED:
    found = 1;
  break;
  case FAIL:
    found = 0;
  break;
}
```

Switch statements are somewhat more flexible than if statements, because each case is **fall-through**. That means that when the MUD finds a matching case statement, it executes that statement and every statement that follows until it finds a line of code marked as **break**.

For the above code, if the result is an AWARD it will display the TM message, and then continue on to the next case statement (SUCCESS). So getting an AWARD gives the TM message and sets the found variable to 1. It then stops, because it hits a break. If the result is SUCCESS, it sets the found variable to 1 and then stops. If it's a FAIL, then it sets the found to 0.

We can add a general catch-all to a switch by adding a special default case. We can use this to deal with results we didn't anticipate.

```
switch (success) {
  case AWARD:
    tell_object (this_player(), "%^YELLOW%^%^BOLD%^You feel a little more "
      "perceptive.\n%^RESET%^");
  case SUCCEED:
    found = 1;
  break;
  case FAIL:
    found = 0;
  break;
  default:
    tell_creator ("drakkos", "Yeah, I don't know what happened here.\n");
}
```

Aside from this new structure, the code should be fairly self explanatory – if the skill check succeeds, the player gets a shiny Genuan cent. If it failed, they get nothing.

We're almost there – at the moment you can search this rock as many times as you like, finding a cent each time if you pass the skill check. This is a rock, not a Northern Rock (teehee). We should make it so that we can only find the coin once per reset.

Scoping Things Out

So, how do we do that? The obvious thought is to use a variable – something like found, in fact. How about if we just put a check at the top to see if found has been set to 1... will that work?

```
int search_rock() {
  int success;
  int found;

  if (found == 1) {
    tell_object (this_player(), "It looks like the rock has already "
      "been searched.\n");
    return 1;
  }

  success = TASKER->perform_task (this_player(), "other.perception", 100,
    TM_FIXED);

  switch (success) {
    case AWARD:
      tell_object (this_player(), "%^YELLOW%^%^BOLD%^You feel a little more "
        "perceptive.\n%^RESET%^");
    case SUCCEED:
      found = 1;
    break;
    case FAIL:
```

```
      found = 0;
    break;
    default:
       tell_creator ("drakkos", "Yeah, I don't know what happened here.\n");
  }

  if (found) {
    tell_object (this_player(), "You have found a shiny Genuan cent!\n");
    this_player()->adjust_money (1, "Genuan cent");
  }
  else {
    tell_object (this_player(), "You don't find anything in the rock.\n");
  }

  return 1;
}
```

Alas, it turns out no. The code doesn't seem to do anything. Why is that?

The answer lies in a programming concept called **scope**. Every variable that is created takes up memory on the computer in which the program is running. This is true regardless of whether the code is on a MUD or on your own computer.

In order to ensure that this memory is made available when you are finished with it, the computer frees up the memory it has allocated once the variable falls out of scope. Variables that are defined within a function are called **local variables**, and exist only as long as that function is executing. Once your search_rock function has finished executing, the memory location occupied by the found variable is released. Then, when it is searched again, a new variable is setup and starts with the value 0 until it is again set by later code. The scope of the variable is the function in which it is defined. That also means that you cannot make use of that variable in other functions.

We can move a variable declaration to the start of the object itself, after the inherit and before the setup:

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

inherit "/std/room/outside";

int found;

void setup() {
```

This increases the scope of the variable to be **class-wide**. It is available to all functions, and persists as long as the object is loaded. Every function can change the state of the variable, but that value the variable gets persists as long as the object is loaded. Sounds like just what we need!

Update your room and search the rock – find the cent the first time, and search again – you'll get the message that indicates the rock has already been searched. A win!

The last part of getting this working properly is to reset the value of found every time reset is called. That bit, at least, is easy:

```
void reset() {
  ::reset();
  found = 0;
}
```

You can test this quite easily – update, search the rock, search it again and get the 'already searched' message. Then, call reset on the room manually:

```
call reset() here
```

And search once more. You'll find the cent again, as if the proper reset period has passed. Pretty nifty, eh?

Conclusion

We're starting to pick up speed in our discussion of LPC – in this chapter we've learned about movement zones, the taskmaster, search add_items, switch statements and variable scope. That's a lot to digest, and you may want to step away from the tutorials at this point to allow the information to sink it. It's a good idea to practice with all of this – try setting up other searchable items in other rooms. Practise is the way to gain understanding, after all.

You should be feeling quite proud of yourself at this point – you're starting to add some fairly sophisticated behaviour to your rooms, and your capabilities are only going to grow as we continue.

Now That We're An Item

Introduction

So, we've got an NPC, and we've got some rooms – the next thing we need to learn how to develop is an item. This is somewhat more complicated than developing either of the others because of the sheer variety of items that can exist – clothes, weapons, armours, backpacks – all of them are created using different inherits and code. There are some commonalities to be sure, but they each have their own little quirks that you need to learn.

More than this, there are two entirely different ways of creating items – one way is a variation of what we've done before – we write an object, and clone it when we need it. The other way is using the MUD's virtual item system. We'll discuss both of these in the course of this document.

Virtually Self-Explanatory

In the long description we wrote for Captain Beefy, we mentioned his deep paranoia about players stealing his shoes – we have not, however, provided him with any. That's because we're going to write a unique pair of shoes for Beefy. The won't do anything special, they'll just have a unique description.

Every object that is loaded on the MUD takes up memory on the system – because there are So Many Items carried by So Many NPCs and Players, there's a huge performance gain to be had by reducing the memory required for these items. The Virtual Item system was introduced to reduce the memory load on poor A'Tuin.

Remember how we talked briefly about the idea of a master object from which we create clones? Every .c file that is loaded on the MUD is a master object, and a master object comes with a memory burden. Virtual objects are just clones of an already existing master object, with all the functions we'd normally associate with setup (such as setting the name, short, and so on) are handled by the MUD as a series of calls. That may sound confusing, so I'll give an actual example of what this does when we've discussed our first virtual object.

Virtual object code files don't look like normal code files – they have their own syntax, and that can be quite confusing. They also have an extension other than .c, and the extension tells the MUD what kind of object we're working with. The basic extensions you'll be working with are as follows:

Extension	Type of Item	In-Game directory
.clo	Clothing	/obj/clothes
.arm	Armour and Jewellery	/obj/armours and /obj/jewellery
.wep	Weapons	/obj/weapons
.food	Food	/obj/food
.sca	Scabbards	/obj/scabbards

There are more of these, but we're only going to discuss clothing in this section. You should browse the indicated directories for examples of other kinds of virtual objects.

Okay, let's start by creating a new directory in our learnville directory – this one will be called items.

```
mkdir /w/your_name_here/learnville/items
```

And we'll need to add a new entry to our base path.h file in /w/your_name_here/learnville/:

```
#define      ITEMS      PATH + "items/"
```

Now, create this file in your new directory, under the name beefy_boots.clo:

```
::Name::"boots"
::Short::"pair of beefy leather boots"
::Adjective::({ "pair", "of", "beefy" })
::Main_plural::"pairs of beef leather boots"
::Plural::"boots"
::Plural Adjective:: "pairs"
::Long::"This is a pair of extremely beefy leather boots.  A person would "
  "need to be very beefy indeed to wear these!\n"
::Material::"leather"
::Weight::6
::Value::200000
::Type::"boot"
::Setup::2000
::Damage Chance::15
```

That, my young friends, is a virtual object. Although it looks entirely different from the LPC code with which you are slowly becoming accustomed, you should be able to see commonalities. Virtual files come as a list of settings, with the values those settings are to have. We update these virtual files in the same way as we do normal files, and we likewise clone them in the same way – clone a pair of them into your inventory once you've updated them.

From the perspective of the person who has them in their inventory, they are indistinguishable from normal LPC objects. That's because that's exactly what they are – they're just written in a different way, and the MUD creates them in a different, more efficient way.

When you tell the MUD to clone a virtual item, it is the extension of the file that tells it what base object it needs to make a clone of – in the case of clothing, it's /obj/clothing.c. It just takes a copy of this object, which has all the functionality but none of the configuration details, and it takes your virtual file as a template for what it should do with it.

It the goes over each of the lines in the virtual file you gave it. The presence of the double colons gives a set pattern for the MUD to parse – It knows that the whatever comes between the first set of double colons and the second set is the name of the setting it needs to change, and whatever comes after the second set of double colons is the value that setting should have.

When it encounters the setting marked **name**, it knows that it translates that into calling **set_name** on the object it has cloned. The value for the set_name is what follows the second set of double colons. Likewise, when it gets to the setting **short**, it knows to call **set_short**.

Don't copy the next bit of code into your project, it's for explanation only. Essentially cloning this virtual clothing file is the same thing to you doing the following:

```
object ob = clone_object ("/obj/clothing");

ob->set_name ("boots");
ob->set_short ("pair of beefy leather boots");
ob->add_adjective (({"pair", "of", "beefy"}));
ob->set_main_plural ("pairs of  beefy leather boots");
ob->add_plural ("boots");
ob->add_plural_adjective ("pairs");
ob->set_long ("This is a pair of extremely beefy leather boots.  A "
  "person would need to be very beefy indeed to wear these!\n"
ob->set_material ("leather");
ob->set_weight (6);
ob->set_value (20000);
ob->set_type ("boot");
ob->setup_armour (2000);
ob->set_damage_chance (15);
```

It probably won't be obvious at this point why this is a good way to do things – trust me when I say though it saves on the memory the MUD uses, and that's a very good thing. The MUD routinely sits at around two and a half gigabytes of memory usage, and it would be a great deal higher still if it weren't for systems like this to keep the requirements low.

But what does it all mean?

I'm going to assume most of what these functions do is self-explanatory… there are some however that are worth spending a little more time delving into so as to understand how to create effective items.

The weight of an object is measured in weight units. Every item that is to have weight (and every item should) will have to weight a minimum of one weight unit. In real terms, a weight unit translates into 50 grams of weight. In the case of our shoes then, they weight 6 * 50g for a total of 300 grams.

The value of the boots is the value in money units – one point of currency is equivalent to one brass coin. There are four brass coins to an AM Penny, and three brass coins to a Genuan cent. Check *help currency* for a full listing of how valuable various currencies are. Beefy's boots are worth 200000 brass coins, which works out to 500 AM dollars. No wonder he was so afraid people would steal them!

Clothes get damaged as their wearer takes damage, and so they need to have a condition set – it is setup_armour that gives the maximum condition of the item. In this case, the clothing has 2000 condition points. For comparison, the obsidian boots you may have encountered in the game have a condition of 6400.

The last bit, set_damage_chance, sets how much damage the item absorbs. Or more correctly, it sets how much damage the item lets through when it itself is damaged. The following chart (from help set_damage_chance) gives you guidance on what those values should be:

Material	Damage Chance
Cloth	20
Hide	17
Bone	16
Leather	15
Wood	12
Copper	10
Bronze	8

Iron	6
Steel	5
Klatchian Steel	3
Stone	3
Octiron	0

Thus, the lower the damage chance, the better protection the item offers. Be very wary of deviating from these values - our standards exist for a reason.

The directory /obj/clothes is the repository for all the in-game clothing that has been written, and you should consult the appropriate subdirectoy in there for guidance as to what values you should set for your values. Going by what's already there is always the best way to design new items - in truth, you're not going to care about these values very much, so just copy values that have already been approved.

Virtual files are perfect for providing simple behaviour, but they do not offer a facility for more complex behaviour. Essentially anything that involves you adding commands, special defences, or general 'less than usual' functionality. For that, we must rely on a standard .c file. We'll see that in practise when we progress onto LPC For Dummies 2.

Beefy's Boots

Now that we have a pair of delicious beefy boots for our NPC, let's give them to him! Sadly, we can't do this through the armoury. Not yet.

The armoury works only on 'live' code – in general we don't want personal creator code (residing in /w/) to be handled through the armoury. The armoury makes a list (and checks it twice) of all the items in the in game directories (such as /obj/clothes, /obj/armour, and so on) and then a list of all the items available to the domain items directories (/d/forn/items, d/am/items, and so forth). What it doesn't do is make the code in your home directory available, or code that is in non-supported directories (all domain objects have to be in /d/domain/items or a subdirectory, for example).

We're thus going to have to handle the provision of a pair of boots manually. First, we make ourselves a container for the boots at the top of Beefy's setup:

```
object boots;
```

Then we clone the boots using the ITEMS #define we added earlier. First though we'll need to add a #include to our path.h at the top of his file, since we currently don't have one.

```
boots = clone_object (ITEMS + "beefy_boots.clo");
boots->move (this_object());
```

Dest and update Beefy and the room from which he is cloned, and you'll find he's now wearing the lovely boots we created for him!

Bling

Let's have a look at a second kind of item created using the virtual object notation – we're going to give Beefy a ring that was given to him by his Dearly Departed wife. Any player who kills him for his jewellery will thus feel like a Real Bastard.

Jewellery is created according to the same basic system, although the specifics of the setup are slightly different:

```
::Name::"ring"
::Short::"beefy diamond ring"
::Adjective::({"beefy", "diamond"})
::Plural::"jewellery"
::Alias:: "jewellery"
::long::"This is a beautiful golden ring set with a gleaming diamond.  "
  "It smells vaguely of beef.\n"
::Weight::1
::Value::80000
::Property::"shop type", "jewellers"
::Type::"ring"
::Setup::320
::Damage Chance::7
::Material::"gold"
```

Most of this should be fairly obvious by now – it shares most of the settings with clothing. The Property tag here sets the item as being jewellery. This check gets used in a few places - for example, the wizard jewellery blorping spell checks this property to see if it's a valid target for the spell, and other places use it to tell what kind of skills are used to repair, and so forth.

How do people know this is a ring given to him by his dead wife? Well, let's add an engraving to it so that people can read the loving message she left for him. We can do this directly in the virtual file, if we like:

```
::Read Mess:: "From your dead wife.", "neat engraving", "morporkian"
```

The first part is the deeply moving message. The second is how the writing is described when someone reads it, and the third is the language in which the message is written. Reading this ring thus will give the following:

```
> read beefy ring

You read the beefy diamond ring: Written in neat engraving:   From your dead
wife.
```

Tragically moving, isn't it?

It's not usually a good idea to include such a read message in the base file, because as far as possible virtual files should be entirely generic for what they are. If we include it in the base file, then anyone who gets a Beefy Ring gets it inscribed with the beautiful poetry of Beefy's wife.

Instead, we can configure this message after the ring has been cloned – that way the basic ring is defined and usable by anyone, but only Beefy's ring has the engraving.

We make the ring available to Beefy in exactly the same way as we made the boots available. We define an object at the top of his setup:

```
object ring
```

Then we clone and move that ring into him:

```
ring = clone_object (ITEMS + "beefy_ring.arm");  ring->move (this_object());
```

And then we manually add the read message afterwards:

```
ring->add_read_mess ("From your dead wife.", "neat engraving", "morporkian");
```

Almost every item in the game can have a read message attached to it in this way, and it's a nice way to add a little bit of sparkle to otherwise non-specific items.

A Word Of Warning About Items

In order to achieve a measure of consistency across the Disc, there are some strict guidelines about what the maximum acceptable values for various settings. As such, before they go into the game all such items have to be approved by a Central Authority. This makes sure that the little knife you casually give to an NPC isn't inadvertently more powerful than the high-end magical dagger people need to spend hundreds of dollars to obtain.

In general, it's better to make use of the large number of items we already have available than code one from scratch. Obviously there are sometimes very specific purposes for new items to exist – you've coded a new shop, or an area has a thoroughly different feel from the rest of the areas on the Disc and needs an infrastructure of items to support that. New items are always going to be a part of new development, and they should be – having new and interesting things available is part of what drives players to explore new areas.

On the other hand, we also want to make sure that the resources we already have are properly utilised. Spend some time becoming familiar with what the armoury has available before you add your own 'black silk shirt' – the chances are something that meets your needs already exists.

If it doesn't, then write away – it's worth consulting with whoever is responsible for administering the approval of the type of objects you are developing to see what guidance they have for you. They may be able to direct you to resources suitable for what you need, or let you know of current acceptance criteria for new items. Or they may just be able to give you useful advice on how to put them together.

Conclusion

We've looked at two kinds of item here – a pair of boots, and a ring. All virtual objects work on largely the same general principles – a file contains a number of settings and the values those settings have. The MUD works out what kind of base object is needed from the extension the virtual object has. It then clones an instance of the base file and configures it with calls (rather than creating a new master object). As far as anyone using the item is concerned, it works identically to any other object written in LPC.

There is a need for care in developing items on Discworld, for it is very easy to upset the delicate balance we have between the items that exist currently and the new items being introduced. Even comparatively minor changes to the values relative to other items can have unusually large consequences. Your Friendly Item Approval Representative will be able to give you formal guidance on what is, and is not, acceptable.

An Inside Job

Introduction

We've got the skeleton of our outside rooms in place already – in this chapter we're going to look at developing our first inside room – an item shop in which we can make available any further items we develop. As a process, this is a largely identical to creating an outside room, except that we have the extra step of setting up stock. We'll have cause to encounter some new concepts as we go through this chapter, so there's plenty for us to talk about.

We're actually going to put in the base code for two rooms – one is a vanilla room that has no special functionality except for a locked door leading to it. The second is the item shop. On our village map (remember that from chapters two and three?), these will be rooms A and B, and have the following filenames:

Map Key	Filename
A	mysterious_room.c
B	stabby_joe.c

So, with no further ado, Let's Get To It!

The Mysterious Room

Firstly, let's create the template for the mysterious room. Everything we've talked about for our outside rooms works for our inside rooms, but we inherit "/std/room/basic_room" instead of "/std/room/outside". So, for the mysterious room:

```
#include "path.h"

inherit "/std/room/basic_room";

void setup() {
  set_short ("mysterious room");
  add_property ("determinate", "a ");
  set_day_long ("This is a mysterious room during the daytime.  It fairly "
    "reeks of mystery.\n");
  set_night_long ("This is a mysterious room during the nighttime.  The "
```

```
    "oppressive darkness hints mysteriously at mystery!\n");

  add_day_item ("mystery", "Daytime all around, and yet you cannot see it.");
  add_night_item ("mystery", "You can't see the mystery for all the "
    "darkness.");

  room_day_chat ( ({ 120, 240, ({
    "The room emits a sense of mystery.",
     "No doubt about it, this is a mysterious room.",
  })}));

  room_night_chat ( ({ 120, 240, ({
    "Was that a mystery there, glinting in the darkness?",
    "Maybe the mystery is a grue?",
  })}));

  set_light (100);

  add_exit ("south", ROOMS + "market_northwest", "door");
}
```

Notice here that the exit we add is of type 'door' – that means pretty much what you'd expect it to mean – the exit is blocked by a door. We need to match this up with a corresponding exit in market_northwest.c:

```
add_exit ("north", ROOMS + "mysterious_room", "door");
```

We've created a door, but it isn't locked. In order to maintain the mystery of the door, we can use modify_exit to lock it shut. Remember how I said earlier that modify_exit had all sorts of cool and interesting powers? Well, let's take a look at what we can do with our mysterious exit here, to heighten it's mystery!

Modifying Exits

The settings to modify_exit come as a listing of pairs of settings and values. With it, we can change the messages people see when they move through the exit, what they see when they look at the exit, and how it behaves when people try to walk through it. We can even add complicated code handlers to an exit – you can control extremely precisely under what conditions an exit may be taken. If you want it to be impossible to take an exit while you're carrying iron in your inventory, you can do that. If you want to make it so that entry is permissible only on the 25th day of Offle Prime, you can do that too. We won't talk about that in this section, but we will in a later section of LPC for Dummies.

The simplest settings just require some numbers and text to describe what should happen. Let's begin simply – let's change how the door appears when people look at it. In market_northwest's setup, we add the following:

```
modify_exit ("north",
  ({
    "door long", "The door hints at mystery within.  Riches too, most "
      "likely.\n",
  })
);
```

Now, when we look north we'll see:

```
> look north

The door hints at mystery within.  Riches too, most likely.  It is closed.
```

If you go north, and look south however, you see:

```
> look south   It's just the south door.   It is closed.
```

The modify exit thus must go in both rooms to which it applies – that way you can have doors that have one description on one side, and a different one on the other. In mysterious_room thus:

```
modify_exit ("south",
  ({
    "door long", "The door hints at mystery within.  Riches too, most "
      "likely.\n",
  })
);
```

If we want to lock it (and we do), we add the locked setting to our modify_exit – a 1 indicates the door is locked, a 0 indicates it is not. We also need to provide the name of a key that will open the door:

So, in market_northwest:

```
modify_exit ("north",
  ({
    "door long", "The door hints at mystery within.  Riches too, most "
      "likely.\n",
    "locked", 1,
    "key", "Mysterious Room Key",
  })
);
```

mysterious_room will have an identical modify_exit, except that it will modify the south exit rather than the north one. Update our rooms, and voila! The door is locked.

Locked doors can be a pain in the backside for a creator, so you can give yourself the ability to walk through such doors by adding the demon property to yourself:

```
call add_property ("demon", 1) me
```

The key to this door is something we're going to make available in our shop... a key is any object that has a property matching the name we've given to the key in our modify exit. When the player attempts to take the exit, the MUD looks through all the items on that player for anything that unlocks the door. You can see this in action by picking any random item in your inventory and using the following call:

```
call add_property ("Mysterious Room Key", 1) random object
```

Make sure you remove the demon property from yourself though, or you'll just ghost through the door as before:

```
call remove_property ("demon") me
```

Modify_exit lets us do much more than we've done here – we can set a door to automatically lock after we close it with the autolock setting (1 indicating it autolocks, and a 0 indicating it doesn't). We can set a difficulty for people attempting to lockpick the door using the difficulty setting – you give this a value from 1 to 10 The lower the value, the easier the lock is to lockpick.

We can also change the messages people see when objects pass through the door by setting three values in the same manner as we did for Beefy's move message. For example, we could do the following in market_northwest:

```
"enter mess", "$N mysteriously enter$s the room.  How mysterious!",
"exit mess", "$N exit$s through the mysterious door.  What wonders "
  "must be found within?",
"move mess", "You walk through the door, excited by the possibilities ‟
  within!\n",
```

And in the mysterious_room:

```
"enter mess", "$N mysteriously enter$s from the mysterious door.  What "
  "wonders were seen on the other side of that portal?",
"exit mess", "$N exit$s through the mysterious door.",
"move mess", "You leave the room, satisfied its mysteries have been "
  "revealed to you.\n",
```

Enter mess is what people see when someone arrives in a room through the exit. Exit mess is what people see when someone leaves through the exit. Move mess is what gets displayed to the player.

So, we've given the door to our mysterious room enough mystery for now. Let's make the key available to those who may be tempted to explore.

Shop 'Till You Drop

We're going to create an item shop here, which is a shop that sells but does not buy. The inherit we use for this is /std/item_shop. Let's create the basic template for that room, and hook it up to our marketplace:

```
#include "path.h"

inherit "/std/item_shop";

void setup() {
  set_short ("Stabby Joe's Emporium of Wonder");
  add_property ("determinate", "");
  set_long ("This is Stabby Joe's Emporium of Wonder, where he sells "
    "wonderful things.  He also stabs people.\n");
  set_light (100);
  add_exit ("south", ROOMS + "market_northeast", "door");
}
```

And in market_northeast, we need an exit linking back to the shop we're setting up:

```
add_exit ("north", ROOMS + "stabby_joe", "door");
```

Update both rooms, and wander into the shop. You'll find that you can 'list' and 'browse' even though you haven't written any code to do that – all of the code for handling the commerce is provided by the inherit we selected.

We can add stock to the room by including an add_object call in the setup, detailing the armoury name of the item we want to add, and how many of the item should be in stock. The item's internal value will dictate how much the item costs.

Let's add a long sword to the stock, to see this in action:

```
add_object ("long sword", 3);
```

Update and 'list', and you'll find that the shop now has three longswords for sale. Alas, they are on sale for provincial money:

```
The following items are for sale:
  A: a long sword for 12 silver coins (three left).
```

The shop has no idea where in the game it actually is – it could be in Ankh-Morpork, it could be in Genua... it could be a shop that trades only in Genuan money but is found in Lancre Town. We need to help the shop with its identity crisis by telling it what kind of money it should accept, and we do that by adding a place property:

```
add_property ("place", "Genua");
```

Other valid places that can be set:

Place	Currency Used
Lancre	Crowns and Shillings
Genua	Ducats and Livres
Ankh-Morpork	Royals and Dollars
Counterweight Continent	Rhinu and such
Djelibeybi	Talents and Tooni

The capitalisation here is important – you won't get the results you're looking for if your place property doesn't match the area exactly.

Update again with one of these set and you'll see the Currency of the Realm change according to where you tell the shop it may be found. For the purposes of the rest of this tutorial, I'm going to assume you've set the currency to be Genuan.

Let's add something else to our shop, to increase its interest a bit more. Unfortunately, the item shop only works with a restricted subsection of what's available in the armoury – it'll work for weapons, armours, scabbards, clothes, foods, jewellery and a few other things, but it won't automatically allow for objects to be added if they reside in a domain . If we want to pull things out of a specific domain, we can set an 'object domain' to complement the selection available. For example:

```
set_object_domain ("forn");
```

Now, let's add something that's available in the forn domain – specifically, the jack in the box. However, let's add it in a slightly different way by giving the shop a slightly randomised amount of the object for sale. With the longsword, there will always be three of these until the shop stock resets. We can add a bit of randomness to this by using the MUD's random number generator, like so:

```
add_object ("jack in the box", 2 + random (2));
```

The random number generator on Discworld generates a whole number between 0 and the number you give it, not inclusive– in this case, it'll generate either 0 or 1 and add that to the 2 we already have. Our shop thus has either 2 or 3 of these objects available.

Stabby Joe

The MUD won't force you to have a shopkeeper for the shop, but it's a good idea to have one – it adds a sense of immersion that is otherwise lacking. So let's create our second NPC – Stabby Joe! He's going to be our shopkeeper and resident psycho. We're going to give him an add_respond_to_with for his cousin, but we're not going to do anything with it right away. Save him in your chars directory under the name joe.c:

```
void setup() {
  set_name ("joe");
  set_short ("Stabby Joe");
  add_property ("determinate", "");
  add_property ("unique", 1);
  add_adjective ("stabby");
  add_alias ("stabby");
  set_gender (1);
  set_long ("This is Stabby Joe, Learnville's main shopkeeper and "
    "resident psycho.  While generally at peace, he can be riled "
    "into fits of towering rage by making reference to his cousin, "
    "Slicey Pete.\n");
  basic_setup ("human", "thief", 150 + random (100));
  setup_nationality ("/std/nationality/genua", "genua");

  load_chat(10, ({
    1, "' Buy my stuff, or I'll kill you.",
    1, "' Good prices on all my stuff!  If you can find anything cheaper "
      "in all of Learnville, I'll kill you!",
    1, "' Satisfaction guaranteed, or I'll kill you!",
  }));

  load_a_chat(20, ({
    1, "' cut cut cut cut cut cut cut cut!",
    1, "' Gods, that was violent!  I blame... the sea!",
  }));

  add_respond_to_with(
    ({
      "@say",
      ({"slicey"}),
      ({"pete"}),
    }),
    "#cousin_response");
```

```
  request_item ("black leather trousers", 100);
  request_item ("black silk shirt", 100);
  request_item ("black soft-soled boots", 100);

  init_equip();
}
```

Everything here is As We Know It. We also know how to load Stabby Joe – we use a variation of the code we've already seen for Captain Beefy. As such, we can include the following in stabby_joe.c (the item shop) to load him:

```
void reset() {
  ::reset();
  call_out ("after_reset", 3);
}

void after_reset() {
  object stabby;
  stabby = load_object (CHARS + "joe");

  if (stabby && environment (stabby) != this_object()) {
    stabby->move (this_object(), "$N appear$s with a pop.",
      "$N disappear$s with a pop.");
  }
}
```

Update the room, and there's our shopkeeper. The only problem is, he's not actually keeping the shop. He's just in the room. We can kill him, and yet still interact with the room perfectly normally. That's not quite what we want – we want the fate of the shop to be tied to the fate of Joe. However, doing that is a discussion for the next section!

Conclusion

In this section of the notes we've explored a little more of the power available in modify_exit, and set up an item shop complete with stock. In the next section, we're going to look at filling out the functionality that's missing here – we're going to need to discuss a new topic in the next section, one that's hopefully going to tie up a lot of what we've been doing into a cohesive whole.

Dysfunctional Behaviour

Introduction

In this chapter we're going to talk about something that will hopefully clarify how everything on Discworld actually fits together. Of a necessity, we have to be quite selective in the theory we cover in this text – we only introduce theory as it impacts on what it is we're trying to do in the examples. As such, there's a lot of 'put this thing here in this way, but don't worry about why'.

Hopefully after you've worked your way through this chapter, a whole lot of what you're doing will make more sense.

Our Task

We've left our shop, and Stabby Joe himself, in a state of incompleteness. We want the shop to be open based on Stabby's presence in the room, and we also want Stabby to fly into a fit of apoplectic rage when his cousin, Slicey Pete is mentioned. Both of those things require us to do more than set a few values in a file – they need us to write actual code.

There's one place already you've written actual code, and that's in the searching of the rock in street_03. Sometimes we want to do something very specific in our areas, and in order to do that we need to go beyond the generic functionality provided by our inherits – for that we need to make use of a programming system called functions. They are also sometimes called methods, and in this chapter we will use the two terms interchangeably.

The Science Bit... Concentrate!

A function is a little section of code that you tag by a name. You've already been writing functions – setup and reset are functions, as is after_reset and search_rock. These are functions that you write so that your rooms do what you want them to. At certain points, the MUD will execute the code that goes with your function – in technical jargon, the MUD **calls** your function. When you room is created, setup is **called**. When the MUD wants to reset the internal state of your room, reset is called. You have no control over these two events – if you change the name of your setup function to set_me_up, you'll find the room doesn't work. This is a convention to which you must adhere.

However, after_reset and search_rock are functions that you told the MUD to call. As such, it doesn't matter what name you give them as long as the name is meaningful. For example, with your after_reset you called it like so:

```
call_out ("after_reset", 3);
```

That sent a message to the MUD, telling it to call the after_reset function after three seconds. If you change it thusly:

```
call_out ("create_npc", 3);
```

It would instead call the create_npc method. It's our choice what we call this function – after_reset is just an informal convention.

Likewise, in your add_item for the rock in street_03:

```
add_item ("jagged rock",
  ({
    "long", "This is a jagged rock.",
    "searchable", "#search_rock",
    "position", "on the jagged rock",
    ({"kick", "punch"}), "Ow!  That stung!\n",
  }));
```

The #search_rock string is what tells the MUD that you want to call a function (indicated by the # at the start) and that function is called search_rock. You could change the name of the function to anything you like provided you link the two up properly:

```
add_item ("jagged rock",
  ({
    "long", "This is a jagged rock.",
    "searchable", "#find_penny",
    "position", "on the jagged rock",
    ({"kick", "punch"}), "Ow!  That stung!\n",
  }));
```

And then for your function:

```
int find_penny() {
  ...
}
```

As long as the name of the function you define is the same as the function you tell the MUD to call, everything will be hunky dory. However, the conventions we adopt for naming these functions were adopted for a reason - they make it considerably easier for people to read and maintain code if they know where they are to look for certain pieces of functionality.

The Structure of a Function

A function has a very particular structure to which it must adhere when written.

```
return_type name_of_function (parameter_list) {   }
```

First, it must declare its return type – that's the kind of information that comes out of the function when it's finished. Our search_rock function returns a 1 to indicate success, and a 0 to indicate failure – thus, we must tell the MUD that our function returns an int.

Next comes the name of the function – we've already seen this in action and should be familiar with it by now.

The parameter list is something we haven't encountered yet – it's information that is provided to our function by the code that calls it. Sometimes the MUD provides information for you when you tell it a function is to be called at some point. Other times you have to provide it yourself.

A parameter list consists of pair of variable types and names, separated by a comma. For example, consider a very simple function that takes two whole numbers and adds them together;

```
int add_numbers (int num1, int num2) {
}
```

The braces indicate the ownership of a function – all code that exists between these two braces consist of the code belonging to it. These are the body of the method. Within the body of the method, we cam make use of variables defined in the parameter list just as if they had already been defined and assigned values:

```
int add_number (int num1, int num2) {
  return num1 + num2;
}
```

When the MUD gets to a return statement in the function, it stops executing the code that belongs to to the function and **returns** whatever value was indicated to the code responsible for calling the method.

Some functions don't return any value, and we indicate those as being of return type **void** (like setup and reset). They require no return statement in the body of their code, and if you wish to terminate them before executing all of the code, you can use a return statement by itself:

```
return;
```

We add functions to our code whenever we want to make a certain piece of code repeatable – rather than copying and pasting the code, we create a function. This means that if the code is broken in one part of the program, we don't need to fix it in all the other places we pasted it – we fix it in the function, and it's fixed everywhere.

Finally, when we want to make use of a function, we simply tell the MUD to call it by giving its name, and any parameters we wish to send it:

```
add_number (4, 5);
```

Does that look vaguely familiar? It should – it's how you've been setting up all of your objects so far – with a series of function calls in which you provide parameters to functions that have already been written.

This function call won't do anything though, because although it will add the two numbers together, the sum of these numbers never get stored anywhere. In the same way that you need an object variable to hold a pair of trousers on the MUD, you need an integer variable to hold the returned value of your function:

```
int num;
num = add_number (4,5);
```

At the the end of this, the variable num will have the value 9.

This should hopefully be making it clearer what's happening in a lot of your code – while we haven't done a lot with dealing with returned variables, pretty much everything we've been doing has been through function calls. When we first setup the equipment for Captain Beefy, we went directly through the armoury which required us to hold the objects in a variable - in technical terms, we got an object reference returned to us and we needed to store that so we could manipulate it. Everything you have done so far has been built on the use of functions. Functions are the engine that drive the MUD.

A Little Bit More...

Okay, just a little bit more theory and then we'll actually go on to do something with these functions. Functions on Discworld are actually broken up into three types.

The first of these are local functions, or **lfuns**. They are functions that are defined inside a game object. For example, add_property is a function that is defined in /std/basic/property.c, and any object that inherits that file will have access to add_property. Objects that don't inherit that file will no have access to that method. Luckily the objects we're creating all have that handled for you, so properties are available to all your rooms, NPCs and items.

Local functions can be called directly using the call command – we've done this a few times as a creator. Call lets us manually call a function, providing the parameters as we do so. The returned value of a call is displayed to us, but we can't do anything with it. They can also be called in other objects using the **->** operator, such as when we do something like:

```
beefy->do_command ("bing madly");
```

The **->** operator tells the MUD 'call the local function do_command on the object referenced with the name beefy'.

The second type of function are external functions, or **efuns**. These are hard-coded into the driver and are available to all objects regardless of what they inherit. You cannot call these functions with the call command, but you can make use of them with the exec command. They do not get used on objects with the -> operator. These cannot be changed easily, as they require a new version of the driver to be developed and installed. The environment() function is an example of such an efun. We call it like this:

```
object env = environment (beefy);
```

We do **not** call it like this:

```
object env = beefy->environment();
```

The last type of function are simulated efuns, or sfuns. While they are not hard-coded into the driver they are defined in a MUD object that makes them available to all of our objects just like an efun. There is no difference to you was a creator between an efun and an sfun – the difference is in terms of performance. Since efuns are defined in the driver, they are executed much more quickly than an equivalent sfun would be. Sfuns can also be changed and added much easier than efuns can, although the set of people who can actually add or change sfuns is fairly limited.

Function Scope

Like variables, functions have a scope – you can't directly make use of a function that is defined in an object other than your current object. You need to use the special arrow notation we've seen – for example, on our ARMOURY:

```
ARMOURY->request_item ("pirate trousers", 100);
```

ARMOURY is an object that is defined in armoury.h, and what this code is telling the MUD is 'call the method request_item on the object defined as ARMOURY'.

There is a secondary syntax for this using an efun called call_other. Call other works like this:

```
call_other (beefy, "do_command", "say Cor, this is cool.");
```

Functionally, this is identical to the following syntax:

```
beefy->do_command ("say Cor, this is cool.");
```

The main benefit is that since the function to be called through call_other is defined as a string, we can actually have variable method calls:

```
string str = "do_command";
call_other (beefy, str, "say Cor, this is cool.");
```

I don't recommend you actually do this, but you may see it in use throughout the code you read and you should know what's happening when you see it.

Onwards and Upwards!

Okay, now we've gotten that out of the way let's incorporate a few functions into our new objects. First of all, let's define what Stabby Joe is going to do when you mention his cousin. Firstly, we're going to keep an internal 'anger counter' for Stabby. This is a number against which we'll roll a random number. If the number we roll is less than his anger, he's angry! If it's more, then he's not quite so angry.

Players mentioning his cousin will receive one of two responses – violence or a warning. The exact response a player will receive will depend on the anger check.

First of all, let's write a stub function – a stub is a function that has no real code in it, but is there so that our object will actually compile when we refer to it:

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

We're going to tell add_respond_to_with that it should use a function to determine how it responds to people mentioning his cousin – we do this by giving the name of the function as a response, with a # symbol at the start of it.

```
add_respond_to_with(
  ({
    "@say",
    ({"slicey"}),
    ({"pete"}),
  }),
  "#cousin_response");
```

So, whenever this response is triggered, it will call the method cousin_response. Be aware that you can't use this syntax for everything – for example, the following *will not work*:

```
set_long ("#random_long");
```

It only works in specific cases, and you should consult the documentation on the function you are using to see whether or not this syntax is supported. It does work as a chat in load_chat and load_a_chat though, which is very useful.

Anyway, back to cousin_response – we add in a stub for this to make him say what he did before. We can't use the apostrophe notation for this, but there's a function called do_command that lets you control your NPC just like you were typing the commands in for him For example:

```
call do_command ("dance") joe
```

This gives the following response:

```
*** function on 'Stabby Joe' found in /obj/monster ***
Stabby Joe dances the disco duck.   Returned: -2631
```

We can make good use of that in any function we write to make Joe do exactly what we want him to do:

```
void cousin_response() {
  do_command ("' Don't mention my cousin!");
}
```

We put these in as stubs first to ensure that everything is working properly – we need to ensure that the connection between our functions is set up before we start writing any more complicated code. First we check the link between our add_respond_to_with and our cousin_response function. We can easily see this simply by mentioning the name to Joe to see if he responds:

```
> You say: slicey pete
Stabby Joe exclaims: Don't mention my cousin!
```

Right, that's working – so now we link up cousin_response and anger_check to see if they connect up properly:

```
void cousin_response() {
  if (anger_check() == 1) {
    do_command ("' I KILL YOU!");
    do_command ("kill " + file_name (this_player()));
  }
  else {
    do_command ("' Don't mention my cousin!");
  }
}

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

Note the way in which we're handling the killin', using the file_name efun:

```
do_command ("kill " + file_name (this_player()));
```

The file_name of an object is an unambiguous unique reference to the object. Find our your own with the following command:

```
exec return file_name (this_player());
```

Then, using the reference it gives you, try interacting with yourself. For example, if your reference was /global/lord#3828617 (which mine was), then try:

```
smile /global/lord#3828617
You smile at yourself.
```

This means we don't need to worry about targeting specific objects by their names – we target them by their references. This is important, it solves several weird issues with targeting and means that your NPC can genuinely differentiate between two objects with the same name.

Anyway, update Joe, and... wait, what's this?

```
/d/learning/learnville/chapter_09/chars/joe.c line 66: Undefined function
anger_check around  == 1) {

*Error in loading object '/d/learning/learnville/chapter_09/chars/joe'
Object: /secure/cmds/creator/upd_ate at line 62
```

What does it mean, undefined function? We've defined it right there!

Actually, this is a layover from the C programming language upon which LPC is based – the MUD reads in our file top to bottom, and it gets to the anger_check function call before it gets to the definition of the function itself. It panics, and halts the process saying 'Hang on! I don't know anything about this anger_check to which you refer! ABORT, ABORT!'

We can simply swap the two functions around to solve the problem:

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

void cousin_response() {
  if (anger_check() == 1) {
    do_command ("' I KILL YOU!");
    do_command ("kill " + file_name (this_player()));
  }
  else {
    do_command ("' Don't mention my cousin!");
  }
}
```

That's an inelegant solution though – you shouldn't have to dramatically alter the layout of your code just to make it run. There's a better solution that involves just telling the MUD 'Don't worry, this function is defined later in the program'. It's called function prototyping.

At the top of your code, just after the inherit (but not before, or you'll introduce a new error) you simply put the definition of the function – its return type, its name, and the parameters, like so:

```
int anger_check();
```

This is the function definition, and as long as LPC sees that before it gets to the function call, it will be happy. There's no need to restructure your code around such an error, just add a function prototype.

Now when we mention Joe's cousin, he says he'll kill us. Progress is being made!

Violence Begets Violence

So, every time someone mentions Pete's cousin, we want to increase our anger counter. For that, we first need an anger counter!

```
void cousin_response() {
  int anger_counter;
  anger_counter = anger_counter + 1;

  if (anger_check() == 1) {
    do_command ("' I KILL YOU!");
    do_command ("kill " + file_name (this_player()));
  }
  else {
    do_command ("' Don't mention my cousin!");
  }
}
```

And then in anger_check, we roll a random number against that anger_counter to see if he's angry:

```
int anger_check() {
  if (random (100) < anger_counter) {
    return 1;
  }
  return 0;
}
```

However, when we update Joe the MUD will once again complain. It's our old friend, the scope problem. Because anger_counter is defined locally in cousin_response, anger_check doesn't have access to it. We could solve this by passing it as a parameter. First we'd need to change our function prototype:

```
int anger_check(int);
```

Then hook up the functions in the new way:

```
void cousin_response() {
  int anger_counter;
  anger_counter = anger_counter + 1;

  if (anger_check(anger_counter) == 1) {
    do_command ("' I KILL YOU!");
    do_command ("kill " + file_name (this_player()));
  }
  else {
    do_command ("' Don't mention my cousin!");
  }
}

int anger_check(int anger_counter) {
  if (random (100) < anger_counter) {
    return 1;
  }
  return 0;
}
```

That solves our scope problem, but it doesn't solve our persistence problem – just like with searching the rock, if we create the variable locally it gets recreated every time the function is called – anger counter will never increase beyond one. We solve the problem then by having the code linked up in the original way, but make anger_counter a class wide variable instead:

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

inherit "/obj/monster";

int anger_check();

int anger_counter;

void setup() {
  // Code for setting him up in here.
}

void cousin_response() {
```

```
  anger_counter = anger_counter + 1;

  if (anger_check() == 1) {
    do_command ("' I KILL YOU!");
    do_command ("kill " + file_name (this_player()));
  }
  else {
    do_command ("' Don't mention my cousin!");
  }
}

int anger_check() {
  if (random (100) < anger_counter) {
    return 1;
  }
  return 0;
}
```

We won't add a reset for anger_counter – Joe just gets angrier, he never calms down over time.

It can take a while before we can tell whether Joe is angry enough to strike, so let's add another function that we can call to tell us his internal state, just to make sure:

```
call query_anger() joe
*** function on 'Stabby Joe' found in
  /d/learning/learnville/chapter_09/chars/joe
***   Returned: 0

> say slicey pete
You say: slicey pete

> Stabby Joe exclaims: Don't mention my cousin!

call query_anger() joe
*** function on 'Stabby Joe' found in
  /d/learning/learnville/chapter_09/chars/joe
***   Returned: 1

> say slicey pete
You say: slicey pete

> Stabby Joe exclaims: Don't mention my cousin!

call query_anger() joe
*** function on 'Stabby Joe' found in
  /d/learning/learnville/chapter_09/chars/joe
***   Returned: 2
```

His anger is increasing nicely! Let's make it easier to test him by allowing ourselves a way to set his anger high enough so that we don't need to repeat the same thing over and over again:

```
void set_anger (int val) {
  anger_counter = val;
}
```

Now set it to something mid-way, such as fifty, and try saying the name a few times:

```
call query_anger() joe
*** function on 'Stabby Joe' found in
  /d/learning/learnville/chapter_09/chars/joe
***   Returned: 52

> say slicey pete
You say: slicey pete

> Stabby Joe exclaims: I KILL YOU!
Stabby Joe moves aggressively towards you!
```

That's one psycho shopkeeper, right there!

A Local Shop For Local People

Once Joe is dead, his shop is still usable. We need to provide a function here to set whether or not the shop is open, and this is done in a slightly different way. In the setup for the shop, add the following:

```
set_open_function ( (: check_is_open :));
```

This is a new kind of syntax for setting a function, and one that is both much more powerful than the # code, and much more problematic. It's called a function pointer. In a section of LPC For Dummies 2, we'll talk about these in more depth, but for now treat it with caution.

Whenever anyone tries to interact with this shop, it'll check the check_is_open function – if that returns a 1, the shop is open. If it returns a 0, the shop is not. Thus:

```
int check_is_open() {
  if (!stabby) {
    return 0;
  }
  else if (environment (stabby) != this_object()) {
    return 0;
  }
  return 1;
}
```

You'll need to add a function prototype for check_is_open:

```
int check_is_open();
```

You'll also need to make your stabby variable of class-wide scope rather than local to after_reset, otherwise the reference is inaccessible inside check_is_open What you'll get from this though is a shop that cannot be interacted with if the shopkeeper is not present. We could improve the function a bit more, such as making it so the shop is also considered closed if he's in the middle of a fight, but that's left as an exercise for the interested reader.

Conclusion

This chapter is somewhat 'think heavy' because of the need for us to talk about how functions work – they are critical to gaining any real understanding of how bits of the MUD communicates with other bits, and so it's worth our while talking about them properly. Hopefully by this point you've now gained a fairly strong understanding of why the magic words you're typing actually make the MUD do interesting things!

We're not done yet of course, there's plenty more for us to talk about before we reach the end of LPC for Dummies one!

Going Loopy

Introduction

Let's move on to the next of the inside rooms in our area – the village pub known as the 'Bitter Pill' and its owner, 'Grumpy Al'. Grumpy Al is a retired wizard from the Unseen University, and in writing his code we'll have cause to look at how to add more interesting combat options to an NPC. We're not going to stress that point at this time, we'll explore interesting combat more as part of LPC For Dummies 2.

So, let's not spend all day standing around and chatting – let's get to work! Our players aren't going to kill themselves, they need us to put the killing machines in for them.

A Basic Pub

Our pub fits into position C on our village map, and will have the filename bitter_pill.c:

```
#include "path.h"
#include <shops/pub_shop.h>

inherit "/std/shops/pub_shop";

void setup() {
  set_short ("Bitter Pill");
  add_property("determinate", "");
  set_long ("This is the only pub to be found for miles - the "
    "Bitter Pill.  They call it that because the prices are "
    "somewhat hard to swallow.\n");
  add_property( "place", "Genua" );
  set_language ("morporkian");
  set_light (75);
  add_exit ("west", ROOMS + "market_northeast", "door");
}
```

This is all familiar territory except for the call to set_language – it's this value that sets what language the menu will be written in. Don't forget it, or you'll get a runtime error any time you try to read the menu.

We'll also need the exit heading back here from market_northeast:

```
add_exit ("east", ROOMS + "bitter_pill", "door");
```

Much like with an item shop, pubs need to be populated with items that are for sale - /obj/food contains a fairly substantial selection of edible and quaffable delights with which to populate our menu. We use the add_menu_item function to provide choices. This function has a substantial number of parameters that go with it, so we'll go over these one by one. For a first example, we're going to make a chicken sandwich available on the menu:

```
add_menu_item ("delicious chicken sandwich", PUB_MAINCOURSE, 2000, "chicken
sandwich");
```

The first parameter is the name the item will have on the menu when players read it. The second (PUB_MAINCOURSE, a define that is to be found in /include/shops/pub_shop.h) is the category under which the item will be displayed, The cost comes next, and the fourth parameter is where the food may be found. If we provide a name like this, it calls upon the armoury. We can also provide a full path name. For a pub, set_object_domain does not work.

Update our room, and read the menu – you'll find the sandwich listed there are the extortionate price we set. No wonder this place is called the Bitter Pill!

Food items are fairly straightforward to setup. Drinks require a little extra configuration because they also need a container – you can provide it without one, but you have to drink quickly before it dribbles through your fingers.

We use add_menu_item as before, except we add a few extra parameters:

```
add_menu_item ("refreshing ale", PUB_ALCOHOL, 1500, "beer", PUB_STD_PINT, 0,
1);
```

The first four parameters work the same way as with the sandwich – the next parameter (PUB_STD_PINT) tells the pub where it can find the container into which these beer should be placed (again, defined in the include file for the pub). The next parameter lets you set how much liquid is going to be in the glass. We use 0 because we're not going to override the standard functionality at all. The last parameter is how intoxicating, on a scale from 1 to 10, the drink should be.

Grumpy Al

Now that we have a pub, let's add its proprietor, Grumpy Al. We'll do his skeleton first, and then the Cool Stuff later:

```
#include "path.h"

inherit "/obj/monster";

void setup() {
  set_name ("al");
  set_short ("Grumpy Al");
  add_property ("determinate", "");
  add_property ("unique", 1);
  add_adjective ("grumpy");
  add_alias ("grumpy");
  set_gender (1);

  set_long ("Grumpy Al is an ex-wizard, in that he no longer studies at "
    "Unseen University.  However, his love of food and of petty "
    "bureaucratic politics has not departed.  Thus, having no effective "
    "outlet to enjoy either of his passions, he has just become grumpy.\n");
  basic_setup ("human", "wizard", 150 + random (100));
  setup_nationality ("/std/nationality/genua", "genua");

  load_chat(10, ({
    1, "' Oh, go away.",
    1, "' Why can't you leave me in peace?",
    1, ({500,
      "' I wrote a nastily worded memo to Stabby the other day.",
      "' He didn't read it.",
      "' I'm not sure he can read.",})
   }));

   load_a_chat(20, ({
     1, "' I'll stick my boot up yer backside!",
     1, "' When I kill you, I'm going to eat you!",
     1, "' Arr num num num!"
   }));

   request_item ("blue silk robe", 100);
   request_item ("black silk underwear", 100);
   request_item ("crooked staff", 100);

   init_equip();
}
```

The only new thing here is one of our load_chats has a more complicated syntax:

```
1, ({500,
  "' I wrote a nastily worded memo to Stabby the other day.",
  "' He didn't read it.",
  "' I'm not sure he can read.",
})
```

This is a story chat – if this chat is selected, Al will make each of the chats one after another, with a delay set by the number we provide. None of the other chats will trigger while the story is being told.

There is nothing else new here, so let's just tie him into our room in the Traditional Fashion:

```
#include "path.h"
#include <shops/pub_shop.h>

inherit "/std/shops/pub_shop";

object al;

void setup() {
  // Usual setup code
}

void reset() {
  call_out ("after_reset", 3);
}

void after_reset() {
  al = load_object (CHARS + "grumpy_al");
  if (environment (al) != this_object()) {
    al->move (this_object(), "$N fall$s in from the roof!", "$N disappears "
      "through a door he draws in mid-air.");
    al->do_command (": stands up and dusts himself down.");
  }
}
```

Note that our reset function here doesn't have the odd ::reset() that our others have. That's because there is no reset function in any other object being inherited by this one, and if we try to include that line we will get a syntax error along the lines of the following:

```
/d/learning/learnville/chapter_10/rooms/bitter_pill.c line 35: No such
inherited function ::reset around ;
*Error in loading object
'/d/learning/learnville/chapter_10/rooms/bitter_pill'   Object: /secure/cmds/
creator/upd_ate at line 62
```

Nothing to worry about, and easily fixed – but it's one of the consequences that comes from working with a code base that has been under constant development for over fifteen years – there are inconsistencies in the ways things are done, and the only way to deal with them is to learn what they are, and how to adapt when the code you had working previously in one file doesn't work in the new one.

Grumpy Al Goes Ballistic

Grumpy Al is not going to have a hair trigger, but he is going to have magic spells he can call on. We can add a spell to an NPC using the add_spell function. We first give the name of the spell by which we want to refer to it, and the filename to where the spell resides. Wizard and witch spells reside in /obj/spells, and priest rituals are to be found in /obj/rituals/

NPCs operate under exactly the same constraints for performing skills that players do. They need components (although we can cheat with this to a degree, if we want). They need the appropriate skills, and they need guild points before they can cast.

Let's give Al a simple spell to begin with - Eringyas' Surprising Bouquet. For those of you unfamiliar with the spell, it's the one that allows the wizard to summon a bouquet of flowers.

In Al's setup, we add the spell. We need to give it a name, but it's a name for our own personal use, not necessarily the formal name of the spell:

```
add_spell ("flowers", "/obj/spells/flowers.c", 0);
```

Update Al and then test his magical powers like so:

```
call do_command ("cast flowers") al
```

A couple of seconds will pass, and he'll begin casting the spell. That's pretty cool, but we've just got that spell there to test his magic... it's not something we're going to use in anger. Instead, we're going to give him the fire bunnies spell.

```
add_spell ("bunnies", "/obj/spells/fire_bunny.c", 0);
```

For this spells, he needs components so we're faced with a game design choice:

- Cheat, and give him as many components as he wants when he needs them.
- Play nice, and give him a set number of components when he is set up.

The second option is both nicer and more interesting – it opens up possibilities for strategy that the first option doesn't (pickpocket all his components before you attack him, for example). That's the option we're going to go for, because it also gives us a chance to talk about another kind of programming structure, the loop!

Components

We're going to setup his components in separate function so as to more easily slot it together with what we already have. The function is going to be called setup_components:

```
void setup_components() {  }
```

He'll need a torch, so we'll give him one of those. He also needs something in which he can store his components, and we're going to give him a large satchel for that. Notice here a slightly different syntax for requesting this item:

```
void setup_components() {
        object satchel;
        request_item ("torch", 100);
        satchel = request_item ("large satchel", 100);
}
```

This matches the syntax we used to get hold of Beefy's ring in earlier chapter. We're going to put our components into Al's satchel rather than into Al directly. We could make him do this with some do_command statements, but that is inelegant. It's much better if he starts off with his components in the satchel, without us needing to fiddle about with the low level details of how they get there.

We're going to start him off with between eight twelve cured carrots and a torch. That introduces our first problem - how do we give him a random number of items in that range? Well, let's start off simply - let's just give him one cured carrot and put it in the satchel.

The request_item method that we're currently using handles requesting the item from the armoury and then moving it into our NPC. That's not quite what we're wanting to do and if we try to use the syntax with which we're already familiar we'll get unusual behaviour (for one thing, the carrot will not move into the satchel). We need to get the item directly from the armoury for this. First include <armoury.h> at the top of your file. This makes available the ARMOURY define that tells the MUD where to find the armoury. Then:

```
void setup_components() {
  object satchel;
  object carrot;

  request_item ("torch", 100);

  satchel = request_item ("large satchel", 100);
  carrot =  ARMOURY->request_item ("carrot");
  carrot->do_cure();
  carrot->move (satchel);
}
```

You may recognise this syntax from the first way we dressed Captain Beefy. We need to make a call to setup_components in Al to make the mud actually use the code in our function, so put a call just before init_equip:

```
setup_components();
init_equip();
```

And add the function prototype after your inherit:

```
void setup_components();
```

Update Al and his room, and you'll see when you look at him:

```
Grumpy Al is an ex-wizard, in that he no longer studies at Unseen University.
  However, his love of food and of petty bureaucratic politics has not
  departed.  Thus, having no effective outlet to enjoy either of his
  passions, he has just become grumpy.
He is in good shape.
He is standing.
Holding : a crooked staff (left hand and right hand).
Wearing : a blue silk robe, a lightable torch and a large satchel.
```

To check what he has inside his satchel, we can use the following command:

```
inv satchel in al
```

Which should give something like the following:

```
Inv of large satchel in Grumpy Al:
  cured carrot (/obj/food/vegetables/carrot.food#6456174)
```

That's exciting! Now, let's give him eight carrots. We'll worry about the random number of carrots later.

The first way of doing this is obvious:

```
void setup_components() {
  object satchel;
  object carrot1, carrot2, carrot3, carrot4, carrot5, carrot6;
  object carrot7, carrot8;

  request_item ("torch", 100);
  satchel = request_item ("large satchel", 100);

  carrot1 = ARMOURY->request_item ("carrot");
  carrot1->do_cure();
  carrot1->move (satchel);

  carrot2 = ARMOURY->request_item ("carrot");
  carrot2->do_cure();
  carrot2->move (satchel);

  carrot3 = ARMOURY->request_item ("carrot");
  carrot3->do_cure();
  carrot3->move (satchel);

  carrot4 = ARMOURY->request_item ("carrot");
  carrot4->do_cure();
  carrot4->move (satchel);

  carrot5 = ARMOURY->request_item ("carrot");
  carrot5->do_cure();
  carrot5->move (satchel);

  carrot6 = ARMOURY->request_item ("carrot");
  carrot6->do_cure();
  carrot6->move (satchel);

  carrot7= ARMOURY->request_item ("carrot");
  carrot7>do_cure();
  carrot7>move (satchel);

  carrot8 = ARMOURY->request_item ("carrot");
  carrot8->do_cure();
  carrot8->move (satchel);
}
```

Copy and paste the code seven times! It'll work - but what if instead of carrots we wanted to give him cured eyes? We need to change the code in eight places. That may seem like a reasonable solution to you, but what if we wanted to give him a hundred chicken feathers? No sir, that dog won't hunt!

Additionally, when you copy and paste you run the risk of introduction what are known as transcription errors - you can lose count and provide more or less carrots than you wanted. Or when making the same modification to multiple lines of repeated code, your attention will wander and you'll introduce typos. Really, this is the kind of thing the MUD should do for us.

In actual fact, it can - there's a special kind of programming structure called the loop that lets us repeat a section of code several times. Loops come in three flavours - the while loop, the do while loop, and the for loop. We'll talk about all three of them, although we only need one for this particular scenario.

Loops

When we want to repeat a piece of code a multiple number of times, we provide the MUD with a loop to make it do the actual work for us. The simplest of these loops is called the while loop, and it will repeat code until a particular continuation condition is no longer met. For example, imagine the following:

```
// Declaration of counter variable.
int num_carrots = 0;
// Setting of the continuation condition.
while (num_carrots < 8) {
  // Upkeep of counter variable.
  num_carrots = num_carrots + 1;
}
```

The continuation condition works just like the condition in an if statement. The difference is that the while loop will keep doing the code in the braces until the condition is no longer met. In this case, it will increase the num_carrots variable to 8 and then it will stop looping (because 8 is not less than 8).

That seems exactly what we want, so let's try that in Grumpy Al:

```
void setup_components() {
  object satchel;
  object carrot;
  int num_carrots = 0;

  request_item ("torch", 100);
  satchel = request_item ("large satchel", 100);

  while (num_carrots < 8) {
    carrot = ARMOURY->request_item ("carrot");
    carrot->do_cure();
    carrot->move (satchel);
    num_carrots = num_carrots + 1;
  }
}
```

Now, let's talk a little about what's happening in the while loop, because it may not be exactly what you may think. Notice here we have one variable called carrot, and each time around the loop we're using that variable. This is often a confusing point when working with code, and so I'll spend a little time explaining what's actually happening.

When you assign an item to an object variable, you're not actually working with the item on the MUD. Your variable is just a shortcut to an item that exists elsewhere in the memory representation that the driver has of Discworld. When your variable loses its scope, the item doesn't stop existing, it just stops being referenced by your own variable. The MUD still knows where it is, what it is, and how it can reference it later.

When you create a carrot here, it clones the object and the MUD itself has a note of that item. You then move it into the satchel, and the satchel gets its own reference to the carrot. If you then assign another carrot to the same variable, it doesn't change the first carrot, or the reference to that carrot in the satchel - you just lose your personal reference to the first one.

As such, we can use the same variable over and over again provided we have no cause to refer to it later. We can't use satchel however for the name of our variable because we need the reference to the satchel later - we have one satchel into which we are putting many carrots - we need that satchel reference so that all carrots go into the same place.

This is a complicated point, and one you can't be expected to fully grasp at the moment. Just keep it at the back of your mind for now.

Update grumpy al and look in his satchel again - you'll see there's one carrot. Hey, what gives?

Much like with virtual objects, the reason behind this is efficiency - rather than store eight carrots, the mud stores one carrot and makes a note to treat it as if there are eight of them. This reduces the load on the MUD and is a Good Thing. You can assure yourself there are eight of them like so:

```
call group_object_count() carrots in satchel in al
```

Externally, as far as you are concerned, there is only one actual carrot object. However, part of the code in that carrot object handles how the carrot should behave, and that code says 'behave as if you were actually eight separate carrots'.

Doing While...

A do while loop works in exactly the same way as a while loop, except that its guaranteed that the code will be executed at least once. If we set num_carrots to 8, then the code belonging to our while loop above would never execute. With a do_while, we can ensure that the code is run once before the continuation condition is checked:

```
do {
  carrot = ARMOURY->request_item ("carrot");
  carrot->do_cure();
  carrot->move (satchel);
  num_carrots = num_carrots + 1;
} while (num_carrots < 8);
```

Do while loops have somewhat specialised use - if you can't think of a reason why you would want to do this, it's fine... you don't have to use them unless you actually need to, but when you do need to they are invaluable.

While loops and do while loops are examples of a set of structures called **unbounded loops** - their continuation can be dependant on unknown constraints. Here we're working with numbers, but that's not where they are best used. They are best used in situations where we don't know how many time we're going to loop. For example, consider the following:

```
while (there are enemies in the room) {
  kill_them();
  kill_them_all();
}
```

We don't know when the enemies will leave the room - they may be killed, they may choose to run away - it's something over which we have no control and thus we manage that uncertainty by using a while loop.

For Loops

We can use a for loop to deal with situations in which we know how many times a piece of code will be executed. That doesn't mean that we know when writing the code how many times we have to execute the loop, just that when the MUD gets to the loop, somewhere we have a variable storing how many times it is going to need to loop over the code. This is a **bounded loop**. It has all the parts of our first while loop, just presented in a slightly different syntax:

```
for (num_carrots = 0; num_carrots < 8; num_carrots = num_carrots + 1) {
  carrot = ARMOURY->request_item ("carrot");
  carrot->do_cure(); carrot->move (satchel);
}
```

The for loop handles keeping the counter variable correct for us - we can concentrate purely on the functionality we wish to repeat. It otherwise works like a standard while loop:

```
for (initialisation ; continuation ; upkeep) {
  // code goes here;
}
```

A for loop can also declare a variable as part of its counter inside the loop declaration, neatly encapsulating everything into a single syntactic structure, like so:

```
for (int num_carrots = 0; num_carrots < 10; num_carrots = num_carrots + 1) {

  // Code
}
```

You can also write the upkeep in a more compact fashion:

```
num_carrots++;
```

Or:

```
num_carrots += 1;
```

Both of these lines of code do (almost) the same thing - they increase the value of num_carrots by one. They actually do very subtly different things in the way they increase the value, but that's of no consequence to us at the moment:

```
for (int num_carrots = 0; num_carrots < 8; num_carrots++) {
  carrot = ARMOURY->request_item ("carrot");
  carrot->do_cure();
  carrot->move (satchel);
}
```

Now all that remains to do is make the loop repeat a random number of times, between eight and twelve. Easy to do, we have all the code - we just use random to set the number of carrots we actually need:

```
void setup_components() {
  int num_to_clone;
  object satchel;
  object carrot;

  num_to_clone = 8 + random (5);
  request_item ("torch", 100);
  satchel = request_item ("large satchel", 100);

```

```
  for (int num_carrots = 0; num_carrots < num_to_clone; num_carrots++) {
    carrot = ARMOURY->request_item ("carrot");
    carrot->do_cure();
    carrot->move (satchel);
  }
}
```

Super - our code-fu grows greater by the day! This isn't something that's just easier to do with a loop - it's actually impossible to do it without one.

Fire in the Hole!

Finally, let's make Grumpy Al cast the spell during combat. We use add_combat_action for this along with the function pointer syntax we saw earlier. In his setup, include this:

```
add_combat_action (15, "cast_spell", (: fire_in_the_hole :) );
```

The first parameter indicates the chance the action will occur. The second is a name we have for the action - we don't need to worry about that. The last bit is the function to be called when the action is to be taken. The MUD will automatically choose a valid target for us to concentrate our attention on, so we don't need to worry about that. Our function is going to have two parameters - the first is the object attacking our object (that'll be us), and the second is the target of the attacker. Don't worry too much about this - this is also a topic for LPC For Dummies 2. It's included here just because it's nice to see our NPCs hurling hot death at people.

We'll need a function prototype for this, but the code itself is quite simple:

```
void fire_in_the_hole (object attacker, object target) {
  do_command ("get carrot from satchel");
  do_command ("' Light them up!");
  do_command ("cast bunnies on " + file_name (attacker));
}
```

Update Al and try to unload the hurts - you'll find every now and again he tries to set you alight. Except, he fails horribly because he's not actually a very good wizard. You can see what his skills are by using the following command:

```
playerskills al
```

A word of warning - this command also works on players but they get an inform that it's being done. Don't use it without permission.

Anyway, you'll see that Al is missing practically all of the skills he'd need to successfully cast the spell, so let's beef him up a bit in his setup:

```
add_skill_level ("magic", 200);
```

Update and load him again and check his skills - much beefier! We can add as many combat actions as we want to make him a challenging foe.

This is obviously quite an unsophisticated action - if he casts too many in a row he'll start suffering from misfires and set himself on fire. The core is there though, and we can expand on this as we choose!

Conclusion

In this chapter we've looked at a new programming structure, the loop, and the code we need to add special attacks to an NPC. We're already capable of creating some quite sophisticated areas, but our journey is not done yet. In this next section we're going to look at a somewhat more complicated variable called an array. Learning about and using arrays is the next real peak of your development as a creator, and from that point onwards whole new worlds of development are available to you!

Arrays, You say?

Introduction

In this chapter we are going to discuss one of the most important coding tools that you have available to you as a creator – the array. Arrays are one of the most important concepts to understand if you want to provide genuinely interesting code in your areas, and it's worth reading over this chapter obsessively until it all makes sense. Then, practice, practice, practice until you are entirely comfortable with the topic. I honestly can't stress this enough – an understanding of arrays forms the most important skill of a creator, as all more complex subjects are inextricably linked with it.

We're going to explore arrays in this chapter through the medium of Slicey Pete, Joe's cousin. Slicey Pete is a wandering merchant who roams the streets of Learnville for its meager business opportunities. We'll also continue with the topic in the next chapter. So, with no more ado, on with the show!

Slicey Pete

Pete isn't just a normal NPC – he's a wandering merchant, and thus he uses a different inherit. Beyond that, he works like all of the other NPCs we've developed this far. Let's look at his template:

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

inherit "/obj/peddler";

void setup() {
  set_name ("pete");
  set_short ("Slicey Pete");

  add_property ("determinate", "");
  add_property ("unique", 1);
  add_adjective ("slicey");
  add_alias ("slicey");
  set_gender (1);

  set_long ("This is Slicey Pete, the cousin of Stabby Joe.  While people "
    "often assume the worst because of his name, he's not a violent man.  "
    "He was just born in Slice.\n");

  basic_setup ("human", "warrior", 50 + random (50));
  setup_nationality ("/std/nationality/genua", "genua");
```

```
  add_property ("place", "Genua");
  add_move_zone ("learnville");
  set_move_after (30, 60);

  load_chat(10, ({
    1, "' Stabby's real name is Madeline, you know.",
    1, "' Poor Madeline - he was such a nice boy at one point.",
    1, "' Buy my stuff, or I'll... well, not be happy.",
  }));

  load_a_chat(20, ({
    1, "' No, I'm just from Slice!  Leave me alone!",
    1, "' I think I just had an accident!",
  }));

  add_respond_to_with(
    ({
      "@say",
      ({"stabby"}),
      ({"joe"}),
    }),
    "' He was a nice lad, once.");

  request_item ("black pinstripe trousers", 100);
  request_item ("crisp white shirt", 100);
  request_item ("leather dress shoes", 100);

  init_equip();
}
```

Note that he inherits /obj/peddler rather than /obj/monster – this sets him up as an NPC you can buy things from. At the moment, he has nothing for us, none stock:

```
> list goods of pete

Slicey Pete says: I am afraid I have nothing for sale.
```

We give him stock in the same way we add stock to an item shop – through add_object:

```
add_object ("carrot", 5 + random (5));
```

So far, so simple. Pete however is a very good merchant and offers a wide variety of food for sale. The full list of things he's going to sell is as follows:

- Bananas
- Carrots
- Apples

- Apricots
- Bread
- Melons
- Cream cakes
- Jam rolls
- Meat rolls.

Now, we can add each of those ourselves in a list:

```
add_object ("banana", 5 + random (5));
add_object ("carrot", 5 + random (5));
add_object ("apple", 5 + random (5));
add_object ("apricot", 5 + random (5));
add_object ("bread", 5 + random (5));
add_object ("melon", 5 + random (5));
add_object ("cream cake", 5 + random (5));
add_object ("jam roll", 5 + random (5));
add_object ("meat roll", 5 + random (5));
```

Again, our problem is one of scale – it might be okay if we're working with this number of items, but a problem gets introduced when we're dealing with more. What if you need to scale back his stock so that he only sells one or two of each? You need to change it in each add_object call – not a huge issue for nine objects, but a big deal for ninety. Our code simply doesn't scale up here.

You may be looking at this and thinking 'A loop of some kind would seem to be the answer', and you're entirely correct – the problem is that we don't yet know how to loop in a way that allows for the item to be requested to be changed. Our loops at the moment work by repeating the same good a certain number of times. What we need is a way to use a loop to do a *slightly different* thing each time. We know how to work with numbers, but strings are a different deal.

We now need a new tool – something that lets us define a list of strings and step over each of them. Well, ask and ye shall receive!

We're going to need to load Pete somewhere, so let's load him in market_southeast. The code for this should be familiar to you by now, so I won't repeat it – check Captain Beefy, Stabby Joe or Grumpy Al if you need a refresher.

The Array

Simply stated, arrays are just lists of data. You've been using them all along, although we haven't stressed then. You've encountered an array whenever you've seen a line of code like:

```
({"some", "things", "here"})
```

Arrays are phenomenally useful, and it's fair to say that until you have mastered them you are only paddling around in the shallow end of programming. Let's talk about how they work!

Arrays are just variables, the difference is that they have multiple compartments into which data can be inserted. All the data has to be of the same type. We declare them slightly differently from the variables we have seen so far – they are indicated with a star symbol by the variable's name. To create an array of strings, we'd use the following syntax:

```
string *my_array;
```

We also set it with a value in a slightly different way – to create an empty array, we use the ({ }) notation:

```
my_array = ({ });
```

We need to set the array to have something before we use it, or we'll likely get Terrible Errors.

If we want to set an array up with some starting values, then we use a variation on that syntax:

```
my_array = ({"some", "things", "here"});
```

When this code is interpreted by the MUD, it sets up a variable with three compartments:

Compartment	Contents
0	"some"
1	"things"
2	"here"

The number of the compartment is called its **index**, and the contents are known as the **element**, as in 'Give me the element at index 2'.

We can pull information out of these compartments using a special notation:

```
string bing = my_array[1];
```

This code will pull out the element at index 1 (in the case of our example, it will be the string "things"). If we want to change a value, then we can do that too:

```
my_array[1] = "pirates!";
```

After this line of code is executed, our array will look like this:

Index	Elements
0	"some"
1	"pirates!"
2	"here"

In both cases you must be careful not to access an index that is either larger than the size of the array, or a negative number. If you do, the MUD will complain of an 'array index out of bounds', which just means you tried to access an element that doesn't actually exist.

If we want to add in a fourth element, then we do that by adding another array to our first:

```
my_array += ({ "matey" });
```

This adds the new element to the end of the array:

Index	Elements
0	"some"
1	"pirates!"
2	"here"
3	"matey"

And to remove an element, like so:

```
my_array -= ({"here"});
```

This will delete every instance of the word 'here' in the array:

Index	Elements
0	"some"
1	"pirates!"
2	"matey"

Array Indexing

LPC provides a number of ways in which we can index elements without simply using integers. We can start indexing from the last element by using the < operator within an array index notation. If we want the last element in an array, we can do it like so:

```
string str = my_array[<1];
```

Or if we want the second last:

```
string str = my_array[<2];
```

We can get a range out of an array by using the .. notation, like so:

```
string *sub_array;  sub_array = my_array[1..2];
```

The range notation is inclusive, so with the above code you would get elements 1 and 2 out of it. You can also combine these notations:

```
string *sub_array;  sub_array = my_array[0..<2];
```

The effect of this array index would be to get all of the elements from the first element (indexed by 0) up until the second last element (indexed with <2). You can even leave off the end part of the range:

```
string *sub_array;  sub_array = my_array[1..];
```

This would get all of the elements of the array from the second one onwards.

There's a lot that we can do with this kind of indexing system, but don't worry about it just yet. Just know that when you see it, that's what's happening.

Array management

Arrays have conceptual complexity to go with them, and as such there are a number of 'management' functions that exist to help you manipulate them. Perhaps the most important of these is sizeof, which gives you the number of elements in an array:

```
int num = sizeof (my_array);
```

With our three elements, the variable num will contain the value 3. We can then use this as a basis for array manipulation – after all, to manipulate an array properly we need to know how big it is - that makes it amenable to manipulation with a bounded loop.

Perhaps the next most important task we perform on a day to day basis with an array is to tell if a particular value is contained with. We do that using the member_array function. This take two parameters – one is the value we wish to search for, and the second is the array through which we wish to search. The function returns a -1 if the item is not present, or the index of the first match if it is:

```
int found = member_array ("pirates!", my_array);
```

With this example, found will be equal to 1 because that's where the search text can be found. If we search instead for 'me hearties', found will be set to -1 because that string is not present in the array.

If we want to pull a random element out of the array, we can use the element_of function:

```
string random_element = element_of (my_array);
```

And we can get several random elements out of the array using the elements_of function. The first parameter to this method is the number of elements you want out of it. The second is the array from which you want to extract them.

```
string *some_elements = elements_of (2, my_array);
```

We can randomise the order of the contents of an array, just like a pack of cards, with the shuffle method:

```
my_array = shuffle (my_array);
```

You can also sort the contents of an array into ascending or descending order using the sort_array function. The number you provide to this method indicates the direction of the sorting - 1 indicates ascending order, and -1 indicates descending order.

```
my_array = sort_array (my_array, 1);
```

That should be enough of a start for us to start using them in our code – like with everything, the only way we ever really start to understand code is when we start to use it. So let's begin!

Setting Up Stock, LPC Style

So, let's get back to our merchant and his unwieldy list of stock. We want all of that to be added in as compact a way as possible. Let's start off by creating an array of things to add. First, at the top of setup, we create our array, and a temporary variable to hold things as they come off of the array:

```
string *items;
string temp;
```

Then, we set items to equal our list of items to be created:

```
items = ({"banana", "carrot", "apple", "apricot", "bread", "melon",
  "cream cake", "jam roll", "meat roll"});
```

Now, we need to go over each of these in turn, and add the object. Once again, we return to the idea of incremental development – let's start by doing something easier. First, let's request whatever is at index 0 in our array:

```
temp = items[0];  add_object (temp, 5 + random (5));
```

That's not so bad – what about getting the second item?

```
temp = items[1];
add_object (temp, 5 + random (5));
```

That's not so bad either – we could go all the way up to index 7 adding items in this way:

```
temp = items[0];
add_object (items[0], 5 + random (5));
temp = items[1];
add_object (items[1], 5 + random (5));
temp = items[2];
add_object (items[2], 5 + random (5));
temp = items[3];
add_object (items[3], 5 + random (5));
temp = items[4];
add_object (items[4], 5 + random (5));
temp = items[5];
add_object (items[5], 5 + random (5));
temp = items[6];
add_object (items[6], 5 + random (5));
temp = items[7];
add_object (items[7], 5 + random (5));
```

Hopefully though a better alternative presents itself – using a loop! We can use the counter of a loop to step through each element in an array, like so:

```
for (int i = 0; i < sizeof (items); i++) {
  temp = items[i];
  add_object (temp, 5 + random (5));
}
```

Suddenly all of that condenses down into a single programming structure that is infinitely extensible – add in a new item to the array, and that item becomes available in the stock. That's pretty cool you have to agree!

Let's try it out though – let's add tomatoes to the stock:

```
items = ({"banana", "carrot", "apple", "apricot", "bread", "melon",
  "cream cake", "jam roll", "meat roll", "tomato"});
```

Dest and update Pete, and you'll find his stock is updated without you adding another line of code:

```
Slicey Pete says to you: I have five plump tomatoes for 1,33Gl each.
```

You sure do, Pete! You sure do.

The Foreach Structure

There exists a specific kind of programming structure designed purely to step over each element in an array – it's called foreach, and you'll see it quite a lot as your read over existing code. It's just a neater way of doing what our loop above does – we can dispense with manually pulling each item off of the array and let the code do it for us. A foreach loop starts at the beginning of the array, and ends at the end, and at each stage it moves on one element forwards through the array. It's not good for when we need fine grained control over how we work with an array, or when we need to know how many iterations we have gone over, or when we need to know where in an array we currently are. However, for straight-forward stepping over an array, it would look like this:

```
foreach (temp in items) {
  add_object (temp, 5 + random (5));
}
```

We use the **temp** variable in the array syntax as a condition for the structure – we don't need to manually query the size of the array or pull the variable off of the array. The loop will begin at the very first element, and pull that element off the array and store it in temp. Next time around the loop, it will take the second element and put that into temp, and so on until all of the elements of the array have been stepped over.

This is a very lovely syntactic structure, and a feature that LPC had long before it made its way into more mainstream languages like Java. The sole drawback is that it doesn't give you access to a specific index number – if you need that at any point in your code, you'll need to rely on a standard for loop

You'll find that a combination of for and foreach will be required to make full use of arrays as you progress through your projects – learning which is best for a particular task is something that will come with experience.

Conclusion

Arrays are a complicated topic, and in the next chapter we're going to look at some of the many varied cool things we can do now that we've added them to our programming toolbox. I'm not kidding when I say that this is what promotes you from the shallow end of the programming swimming pool – without even talking about any other data types, we can do practically anything we'd ever need by relying purely on arrays.

There's one more standard data type we're going to introduce in LPC for Dummies, but our journey through introductory syntax is starting to come to a conclusion – we've covered some scary territory over the past chapters and you are to be congratulated for reaching this far! Now, go practice arrays until your brain bleeds!

Hooray for Arrays

Introduction

In the last chapter we looked at the basics of using arrays in LPC – that was a big step, and it opens up a lot of new things we can talk about. Arrays are embedded deeply in the mudlib, to the point that you can't do anything of consequence without working with them in one capacity or another. Most of the useful functions of the MUD return arrays rather than single bits of data, and in this chapter we're going to look at some of these.

In the process, we'll add the last of the rooms to our village, adding in interactivity based on what a player has in their inventory, among other things.

A Secret To Be Discovered

Look back at our map of Learnville – there's one room we still haven't done, marked D on the map. It's connected to market_southeast, but we don't have anything for it yet. The room itself is going to be a general store – that is, a shop that buys what players have to sell. In the game proper, we try not to have too many of these because each one is an additional level of artificiality in an already broken game economy, but every substantial new area usually has one for playability reasons. However, this is no normal general shop – it is a shop full of hungry grues!

We'll start with the usual first step, our skeleton implementation. We'll save this as magic_shop.c:

```
#include "path.h";
inherit "/std/shops/general_shop";

void setup() {
  set_short ("magic general shop");
  add_property ("determinate", "");
  set_day_long ("The fingers of sunlight do not touch this occult "
    "place.  Hope has long deserted this room.\n");
  set_night_long ("In the darkness, it's possible to send occult, "
    "eldritch energies seeking entry.  Anyone in the room at "
    "night without the special magical safety item is likely "
    "to be eaten by a grue.\n");
  set_light (100);

  add_day_item("occult place", "There's a definite sense of occultness "
    "in the air.");
  add_day_item (({"occult", "occultness"}), "Yes, that sort of thing.");
```

```
  add_night_item ("occult eldritch energies", "They are very oblong.");
  add_night_item ("special magical safety item", "Hoo-boy, you better hope "
    "you have that with you!");

  add_property ("place", "Genua");

  room_day_chat ( ({ 120, 240, ({
    "You are probably safe here, for the time being.",
    "This room makes you sad.",
    "You sense that the room doesn't want you to be here.",
  })}));

  room_night_chat ( ({ 120, 240, ({
    "You better hope you have the magic item!",
    "I feel sorry for anyone who doesn't have that special item!",
    "#eaten_by_grue",
  })}));

  add_exit ("west", ROOMS + "market_southeast", "door");
}

void eaten_by_grue() {
  // We'll fill in the code for this later.
}
```

Note that one of the room_night_chat entries is to a function called eaten_by_grue. We'll come back that one soon, but we'll just add in a stub for it now. Stub methods are a great way to structure your code without needing you to have to actually code everything at once.

General shops are unusual in that they require a second room to exist before they work properly – every general store should have a storeroom that players can't get into. It's kind of like a backpack for the shop – it's where everything gets stored. The stock of a general store is entirely dependant on what is in the storeroom – so we need to have whatever stock we're interested in supplying created inside the storeroom. We do that just like we did for making items available within our NPC through the use of request_item. We'll need to restock the shop periodically, so we'll handle that in reset.

Our template for the file, which we will save as magic_storeroom.c:

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

inherit "/std/storeroom";

void setup() {
  set_short( "store room" );
  set_long( "This is a storeroom.  You shouldn't be here. "
    "If you are, leave (or ask a creator to get you out) and "
    "bug report how you got in.\n" );
  add_exit( "south" , GENUA_AVENUE + "general_store" , "path" );
  set_light( 100 );
}
```

```
void reset() {
  // To follow
}
```

And then we need to tell our general store to make use of the storeroom we have provided – in the setup for magic_shop goes the following line of code:

```
set_store_room (ROOMS + "magic_storeroom");
```

Update the room and our general store works like we'd hope – there's no shopkeeper here, and we're not going to add one because this is a magic shop. Sell an item to the shop and you'll find it functions like you'd expect an in-game store to work.

It starts off with no stock – we're going to have to handle the stock generation ourselves, and for that we need arrays.

Item Matching

Our example of an array in practice from the last chapter was somewhat artificial – although we did gain a real improvement in the use of an array, we didn't have to use one. However, it was an instructive example for what we're going to do here.

We want to setup new stock every time reset is called, but we also don't want to overstock items. Imagine the following:

```
void reset() {
  for (int i = 0; i < 2; i++) {
    request_item ("torch", 100);
  }
}
```

Seems fine, right? Reset is called and we provide two lightable torches to the shop. That will work just as we might expect, but over time and resets that stock will grow, and grow, and grow, and grow. There is a point where we just want to be able to say 'stop, don't create any more of these'. Around about eight would be a reasonable number of torches.

In order to do that, we need to know how many torches there are in the room, and we don't yet know how to do that. We could keep track of how many we have created, but that doesn't catch ones that get sold to the shop by players, or bought, or shoplifted, or that otherwise leave the storeroom. Really, we want to be able to say in reset 'Count how many torches there are, and if there are less than eight clone some in'.

The function that lets us count how many items are in a particular container is called match_objects_for_existence, and it returns – yes, you guessed it, an array!

Let's see it in action by finding all the lightable torches in the room:

```
object *torches;
torches = match_objects_for_existence ("lightable torches", this_object());
num_torches = sizeof (torches);
```

Our matching function takes two parameters – one is the string against which we're going to try to match against an object, and the second is the container in which we're going to check (in this case, it's the room itself).

The match_objects_for_existence function makes ambiguous matches, which means that it will match in the same way the game does when you try to interact with items in your inventory. Torches can be held as weapons, and so if you changed the check like so:

```
torches = match_objects_for_existence ("weapons", this_object());
```

You'd still get the torches returned. Note too that we search for a plural rather than a singular – the following would get us one torch out of the lot:

```
torches = match_objects_for_existence ("lightable torch", this_object());
```

Once we have list of items, we get the size of the array – this gives us how many items are in the storeroom. By doing this check, we can limit at what point we stop adding torches to the stock in a reset – people can still sell them to the shop and so the actual stock in the shop may be more than the value we've set. The only thing our code does is make sure that we only add torches if there are currently less than eight available, like so:

```
void reset() {
  object *torches;
  torches = match_objects_for_existence ("lightable torches", this_object());

  num_torches = sizeof (torches);
  if (num_torches < 8) {
    for (int i = 0; i < 2; i++) {
      request_item ("torch", 100);
    }
  }
}
```

We're also going to add another item to the store – a special item we create ourselves – it's the magic item that stops us being eaten by a grue. It's going to be a special amulet – when the eaten_by_grue function is called above, we're going to eat anyone who doesn't have this special item in their possession. Poor souls!

Jewellery has a .arm extension as we have already seen, and ours is going to be called black_box_recorder.arm:

```
::Name::"amulet"
::Short::"black box recorder amulet"
::Adjective::({"black", "box", "recorder"})
::Plural::"jewellery"
::Alias::"jewellery"
::Main Plural::"black box recorder amulets"
::Long::"This is a special necklace that records information about your "
  "course and trajectory just in case you are eaten by a grue.  Grues "
  "like to see these on people, because grues are all in love with "
  "Sarah Nixey.  No-one wearing one of these need fear a grue!\n"
::Type::"necklace"
::Material::"gold"
::Property::"shop type", "jewellers"
::Value::8000
::Damage Chance::4
::Setup::200
::Weight::1
```

And so, in our shop reset, the full function is as follows::

```
void reset() {
  object *torches;
  int num_torches;
  object *amulets;
  int num_amulets;
  object ob;

  torches = match_objects_for_existence ("lightable torches", this_object());
  num_torches = sizeof (torches);

  if (num_torches < 4) {
    for (int i = 0; i < 2; i++) {
      request_item ("torch", 100);
    }
  }

  amulets = match_objects_for_existence ("black box recorder amulets",
    this_object());
  num_amulets = sizeof (amulets);
  if (num_amulets < 1) {
    ob = clone_object (ITEMS + "black_box_recorder.arm");
    ob->move (this_object());
  }
}
```

Now that our shop is setup and we have our special item available, let's deal with people getting eaten by grues!

A Grue Some Fate!

The criteria for being eaten by a grue is fairly simple – if it's night-time and you're not wearing the amulet, you get eaten, arr num num num. This is a fate for everyone in the room, and so our function has to check, for every player and NPC in the room, if they have protection. Let that be a lesson, boys and girls - don't go out without some protection on you.

Let's start by just getting all the objects that are currently in the room. We do this using all_inventory. This will get everything in the room – players, NPCs, and items that happen to be lying on the ground.

```
void eaten_by_grue() {
  object *room_contents;
  room_contents = all_inventory (this_object());
}
```

There are more compact ways to do the next few bits, but don't worry about those – we're going for simple, easily comprehensible code here so as each step is clearly understandable.

Once we have our list of objects, we want to get the ones that are classed by the MUD as living (NPCs and players). We use the living efun for this – if we wanted players only, we'd use the interactive or userp efuns instead:

```
void eaten_by_grue() {
  object *room_contents;
  object *living_objects = ({ });
  object ob;

  room_contents = all_inventory (this_object());

  foreach (ob in room_contents) {
    if (living (ob)) {
      living_objects += ({ ob });
    }
  }
}
```

Now, we want to step over each of those living objects we've found, and detect whether they have the amulet. Let's slow down here and think about this incrementally – how would we do it for one living object?

Well, first we'd need to see whether that object had an amulet on them – that's easy, that's what match_objects_for_existence does:

```
matches = match_objects_for_existence ("black box recorder amulet",
  living_objects[0]);
```

The amulet itself has a query_worn_by function that tells us who is wearing it, so we can use that. We need to check over each of the amulets returned by match_objects_for_existence, even we expect players to just have one. They may have two, and it would be a little unfair if they were eaten because we happened to check if they were wearing the wrong one.

We're going to use a integer variable here called safe to determine if someone is to be eaten. If safe is 1, that individual doesn't get eaten. If it's 0, they do.

```
safe = 0;

matches = match_objects_for_existence ("black box recorder amulet",
  living_objects[0]);

foreach (amulet in matches) {
  if (amulet->query_worn_by() == living_objects[0]) {
    safe = 1;
  }
}
```

We can assume at the end of this foreach that any player it is still dealing with is not protected from our vicious grues. We tell them they have been found wanting, and the unceremoniously slay them:

```
if (safe == 1) {
  tell_object (living_objects[0], "You are judged and found worthy!\n");
}
else {
  tell_object (living_objects[0], "You are devoured by a grue!\n");
  tell_room (this_object(), living_objects[0]->query_short()
    + " is devoured by a grue!\n",  ({ living_objects[0] }));
  living_objects[0]->do_death();
}
```

There's a lot happening here, so let's look at it in context:

```
safe = 0;

matches = match_objects_for_existence ("black box recorder amulet",
  living_objects[0]);
```

```
foreach (amulet in matches) {
  if (amulet->query_worn_by() == living_objects[0]) {
    safe = 1;
  }
}

if (safe == 1) {
  tell_object (living_objects[0], "You are judged and found worthy!\n");
}
else {
  tell_object (living_objects[0], "You are devoured by a grue!\n");
  tell_room (this_object(), living_objects[0]->query_short()
    + " is devoured by a grue!\n", ({ living_objects[0] }));
  living_objects[0]->do_death();
}
```

Please make sure you understand each step and how all the variables relate to each other - it's quite complicated. This is easily the most complicated piece of code we've encountered thus far. However, it provides a baseline for some very common functionality.

Note that this only works for one single individual in the room – our last step is to do all of this in a foreach so it steps over each individual and performs the same checks:

```
foreach (ob in living_objects) {
  safe = 0;

  matches = match_objects_for_existence ("black box recorder amulet",
    ob);

  foreach (amulet in matches) {
    if (amulet->query_worn_by() == ob) {
      safe = 1;
    }
  }

  if (safe == 1) {
    tell_object (ob, "You are judged and found worthy!\n");
  }
  else {
    tell_object (ob, "You are devoured by a grue!\n");
    tell_room (this_object(), ob->query_short()
      + " is devoured by a grue!\n", ({ ob }));
    ob->do_death();
  }
}
```

The tell_room and tell_object functions are new to us – they're what's used to send messages around the MUD. The first tells the player specifically what happened to them:

```
tell_object (ob, "You are devoured by a grue!\n");
```

The first parameter is the object to which we send the message, and the second is the message itself.

Tell_room is a little more complicated – it tells everyone in the specific container object (in this case, a room), except for those specified as being excluded. The first parameter is the room to send the message to. The second is the message itself, and the third is an array of objects that are not to see the message. We exclude the current object from the tell_room, because they already get a tell_object sent to them.

Update the room and call eaten_by_grue to make sure it works – try it while wearing the amulet, and try it without.

When you've tried it without, you'll notice something disappointing – the death message is Very Dull:

```
[drakkos <drakkos> killed by drakkos (with a call)]
```

Jeez, what is the point of being eaten by a grue if you don't at least get to see an interesting death message? No point at all, I say... so let's make sure we have one. Death messages are based on a query_death_reason function defined in the object that is responsible for calling do_death. So let's add one of those:

```
string query_death_reason() {
  return "being eaten by a grue";
}
```

This by itself is not enough, we also need to add the room to the individual's list of attackers so that the MUD knows what caused the death. We use the attack_by method for this, before we call do_death:

```
ob->attack_by (this_object());    ob->do_death();
```

And that, as they say, is it! Next time the grues find you wanting, the following death message will be triggered:

```
[drakkos <drakkos> killed by being eaten by a grue <Learning>]
```

That's much more entertaining, and that's what we're all here for.

Getting To Our Shop Of Horror

At the moment our shop is inaccessible. We need to change that in our market_southeast file. To avoid people simply stumbling into the store with no warning, we're going to make entrance based on an exit function. First, we add the exit as normal to our room:

```
add_exit ("east", ROOMS + "magic_shop", "door");
```

And then we modify that exit – in this case, to indicate a function that is to be called when people attempt to take the exit:

```
modify_exit ("east", ({
  "function", "warn_player"
}));
```

The exit function will only be called if you are not currently sporting the demon property, so make sure you remove that before you attempt to write this part of the room.

Whenever anyone tries to take this exit, the MUD will call the function warn_player. If that function returns a 1, the player is permitted to take the exit. If that function returns a 0, then they are stopped.

A special word of warning – don't use this_player() in an exit function – if a player is following another player through an exit, only the first player will ever be returned from this_player(). Instead, use the parameters passed to the function by the MUD:

```
int warn_player (string direction, object ob, string special_mess) {
}
```

The first parameter is the direction of the exit. The second is the object taking the exit (you use this rather than this_player()), and the third is the message that will be displayed to the destination room. Usually we don't need to worry about either the first or third parameters, and can concentrate purely on the second.

We're going to provide a warning here - when a player is warned, they get a timed property placed on them. If they attempt to take the exit while they have the property, they get let through:

```
int warn_player (string direction, object ob, string special_mess) {
  if (ob->query_property ("been warned about grues") == 1) {
    return 1;
```

```
  }
  tell_object (ob, "The room ahead smells of grues... are you sure you "
    "want to risk it?\n");
  ob->add_property ("been warned about grues", 1, 120);
  return 0;
}
```

However, when taking this exit for the first time, we get an ugly bit out output:

```
The room ahead smells of grues... are you sure you want to risk it?
Try something else.
```

That 'try something else' is a consequence of the way the MUD deals with error messages. We want rid of that, so we need to add a little bit of black magic to our function - a notify_fail function, which can be used to override the normal 'try something else' error message as we see above. This is the only place you should use notify_fail without a very good reason – if you don't know what those reasons might be, then you don't have one.

```
int warn_player (string direction, object ob, string special_mess) {
  if (ob->query_property ("been warned about grues") == 1) {
    return 1;
  }
  tell_object (ob, "The room ahead smells of grues... are you sure you "
    "want to risk it?\n");
  notify_fail ("");
  ob->add_property ("been warned about grues", 1, 120);
  return 0;
}
```

Notify_fail lets you change the fail message that appears, and in this case we simply suppress it entirely - we already give them the information they need as part of the tell_object. Update the room, and everything should fit together like a greased up jigsaw.

Conclusion

Arrays have unlocked some truly powerful functionality for us, but the manipulation of arrays may still be confusing. All I can recommend is practice – it doesn't get any more important than understanding how arrays work, and if you are not one hundred percent on how they work you should find code and read code and ask questions and shout and scream and stamp your feet until you are. I'm not even kidding!

The only bit of syntax left for us in LPC for Dummies one is the mapping, and that's the topic for the next chapter. You've already come so far – there's not long left to go!

Mapping It Out

Introduction

The last data type we're going to explore as part of LPC for Dummies 1 is the mapping. It goes by different names in different programming languages, but if you ever hear people talking about an 'associative array' or a 'hashtable' or 'hashmap', it's this data type to which they are referring. Mappings are an internally complex data type, but you don't need to worry about any of that – we just need to know how to manipulate them, and that's our topic for this section.

So, let's not dilly dally, let's get stuck in!

The Mapping

Imagine you wanted a way to link together pieces of data. For example, if we wanted to store the guild level and the name of players, we'd need two arrays – one of strings containing names, and another of integers holding levels, like so:

```
int *guild_levels = ({ });
string *player_names = ({ });
```

Then, when we wanted to add a player's guild level, we'd add their name and level to the appropriate arrays:

```
player_names += ({ "drakkos" });
guild_levels += ({ 1337 });
```

We'd need to do this each time that we wanted to add a new pairing of data:

Index	player_names	guild_levels
0	"drakkos"	1337
1	"taffyd"	666
2	"terano"	1227

A mapping is essentially a data structure that allows for that kind of relationship between elements. Writing your own association between arrays is time consuming and prone to problems when it comes to making sure everything gets manipulated at the right time at the right way in both arrays, and as soon as one mistake is made with this the entire set of data loses integrity. Using a mapping means that the MUD does all the Tedious Busy Work for you, and that's what we're looking to do as programmers – offload tedious busy work onto the computer.

A mapping is declared thusly:

```
mapping my_map;
```

And set with an empty internal state using the mapping notation:

```
my_map = ([ ]);
```

Note that this is subtly different from an array – it's square brackets within rounded brackets rather than braces within rounded brackets.

If we want to setup a mapping with some starting values, we use the following:

```
my_map = ([
  "drakkos" : 1337,
  "taffyd": 666,
  "terano": 1227,
]);
```

Note that each of the key-value pairs are separated by a colon rather than a comma.

As with the array, we need to know how to manipulate a mapping before we can really make use of it, and before we do that we need to talk about the terminology of a mapping. Unlike an array, a mapping does not have an index by which it can refer to elements. Instead, a mapping has a **key** and **value** pairing. A particular key is linked to a particular value. Our example above, in a mapping, would look like this:

Key	Value
"drakkos"	1337
"taffyd"	666
"terano"	1227

Thus, the key "drakkos" will have the value 1337, and the key "taffyd" will have the value 666.

To put a value into the mapping, we use the following notation:

```
my_map ["key"] = value;
```

So, to add in a fourth entry:

```
my_map ["gruper"] = 69;
```

At this point, our mapping looks like this:

Key	Value
"drakkos"	1337
"taffyd"	666
"terano"	1227
"gruper"	69

To pull out a value, we use a similar notation:

```
int value;
value = my_map["terano"];
```

We don't need to know in what position of the mapping we need to look for values, because the mapping does that for us. Indeed, you can't guarantee that the order in which you add values has any relationship to how they are stored internally for all sorts of Complicated Technical Reasons. You have to think of it a bit like a bag of data rather than a list.

To remove an element from the mapping, we use the map_delete function, like so:

```
map_delete (my_map, "terano");
```

This looks through our keys in the mapping for a match, and then removes that match from the mapping:

Key	Value
"drakkos"	1337
"taffyd"	666
"gruper"	69

Although syntactically simple, they are quite complicated conceptually – so once again, we're going to use an example to illustrate how they actually work.

The Magic Hate Ball

We're going to add a new item to our marketplace – a magic hate ball mounted on a plinth. This will be in market_northeast:

```
#include "path.h"

inherit "/std/room/outside";

int do_hate();

void setup() {
  set_short ("northeast corner of the marketplace");
  add_property ("determinate", "the ");
  set_day_long ("This is the northeast corner of the marketplace.  "
    "A magic hate ball is mounted on a plinth, attached by a chain.  "
    "Sunlight glints on its malevolent surface.\n");
  set_night_long ("This is the northeast corner of the marketplace.  "
    "A magic hate ball is mounted on a plinth, attached by a chain.  "
    "Starlight glints on its malevolent surface.\n");
  add_item ("magic hate ball",
    ({
       "long", "The magic hate ball gives wisdom when you shake it!",
       "shake", (: do_hate :),
    })
  );

  set_light (100);
  add_zone ("learnville");

  add_exit ("north", ROOMS + "stabby_joe", "door");
  add_exit ("south", ROOMS + "market_southeast", "path");
  add_exit ("east", ROOMS + "bitter_pill", "door");
  add_exit ("west", ROOMS + "market_northwest", "path");
  add_exit ("southwest", ROOMS + "market_southwest", "path");

  modify_exit ("southwest", ({"obvious", 0}));
```

```
  set_linker ( ({
    ROOMS + "market_northwest",
    ROOMS + "market_southwest",
    ROOMS + "market_southeast",
  }) );
}

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

Note that we're using the function pointer notation for the add_item – we need to use that when we attach functions to verbs in add_items. In LPC For Dummies 2 it will become clear why, but for now you're going to have to accept that sometimes you can use the # notation, and sometimes you need to use the function pointer notation.

What we're going to get the magic hate ball to do is give people who shake it a venomous message, and if they shake it again it repeats the same message. Thus, everyone gets a random message the first time they shake it, but the ball is consistent with its advice. We need a class-wide variable for this so that it gets stored between executions of our function:

```
mapping previous_responses = ([ ]);
```

In our function, we first need an array of possible responses:

```
string *random_responses = ({
  "Shut up and die.",
  "I slept with your wife.",
  "Suicide is your only option.",
  "Everyone hates you.",
  "You suck.",
});
```

Back to the idea of incremental development – let's just follow through the process of someone getting a response for the first time. We select a random response, and display it:

```
int do_hate() {
  string response;
  string *random_responses = ({
    "Shut up and die.",
    "I slept with your wife.",
    "Suicide is your only option.",
    "Everyone hates you.",
    "You suck.",
  });
```

```
  response = element_of (random_responses);
  tell_object (this_player(), "The magic hate ball says: "
    + response + "\n");
  return 1;
}
```

That will give a random response each time, but if we want to store the response we then need to put it in our mapping. First, we need a key – this will be the name of the player. We declare a string variable called player at the top of our function, and then set it to be whatever the player's name is:

```
player = this_player()->query_name();
```

Before we get to generating a response, we need to check to see if there's been a previous response by querying our mapping with the key we just acquired:

```
response = previous_responses[player];
```

The response we then give to the player is dependant on whether there is anything in this variable, like so:

```
if (!response) {
  response = element_of (random_responses);
  previous_responses[player] = response;

  tell_object (this_player(), "The magic hate ball says: "
    + response + "\n");
}
else {
  tell_object (this_player(), "The magic hate ball says:  I already "
    "told you:  " + response + "\n");
}
```

Putting that all together gives us the following:

```
int do_hate() {
  string response;
  string player;
  string *random_responses = ({
    "Shut up and die.",
    "I slept with your wife.",
    "Suicide is your only option.",
    "Everyone hates you.",
    "You suck.",
```

```
  });

  player = this_player()->query_name();
  response = previous_responses[player];

  if (!response) {
    response = element_of (random_responses);
    previous_responses[player] = response;
    tell_object (this_player(), "The magic hate ball says: "
      + response + "\n");
  }
  else {
    tell_object (this_player(), "The magic hate ball says:  I already "
      "told you:  " + response + "\n");
  }
  return 1;
}
```

Let's step through that process from the perspective of a player. Player comes along and says 'Hey, cool eight ball. I am going to shake it 'till I break it!'. They shake the ball, and the first thing the function does, after setting up the random responses, is to get the player's name, which is "drakkos", and then query our mapping to see if the key "drakkos" has a value associated with it. Our mapping though is completely empty.

It's empty, so no response is to be found. The code then picks a random response from our list, let's say it's "You suck", and then puts that value into the mapping associated with the key of the player name:

Key	Value
"drakkos"	"You suck."

It then prints out our message:

```
> shake ball
The magic hate ball says: You suck.   You shake the magic hate ball.
```

'My word!', exclaims Drakkos. 'That is rather rude, and no mistake! Why, I shall shake this little rascal again to see what it has to say!"

Drakkos shakes the ball again – this time, when it comes time to query the mapping, there's already a response stored. That gets stored in the variable 'response', and when we get to the if statement response is no longer empty and so the else clause is executed:

```
The magic hate ball says:  I already told you:  You suck.
You shake the magic hate ball.
```

Saddened and appalled by this inhuman criticism, Drakkos wanders off into a shop, and ends up eaten by a grue. This area, he decides, was not written by a nice person. Why so mean, magic hate ball? Why so mean?

More Mapping Manipulation, Matey

There are three further things we might like to do with a mapping in order to manipulate it effectively. One is to get a list of all the keys in a mapping – we can do that with the keys function. It returns it as an array:

```
string *names;
names = keys (my_map);
```

This will give us everything in the left hand column of the mapping. Going back to our example mapping:

Key	Value
"drakkos"	1337
"taffyd"	666
"terano"	1227
"gruper"	69

Getting the keys of this will return the following array:

```
({"drakkos", "taffyd", "terano", "gruper"})
```

We cannot guarantee that the elements will be returned in that particular order, but all of those elements will be present.

Another thing we might like to be able to do is get all of the values. This is done in a similar way, using the values function:

```
int *levels;
levels = values (my_map);
```

This will return the following array from our example mapping:

```
({ 1337, 666, 1227, 69 })
```

Again, in no guaranteed order. If we want to enforce some kind of order on this, the we must handle it ourselves.

Finally, we may wish to step over each of the key and value pairings in turn, performing some kind of operation. We can do that longhand like so:

```
string *keys;
string key, value;
keys = keys (previous_responses);

foreach (key in keys) {
  value = previous_responses[key];
  tell_object (this_player(), key + " was told '" + value + "'\n");
}
```

However, LPC is nice enough to give us a foreach structure purely for handling mappings! The above code rolls into a simple foreach, like so:

```
foreach (key, value in previous_responses) {
  tell_object (this_player(), key + " was told '" + value + "'\n");
}
```

Let's try that out by adding a 'consult' option to our ball:

```
add_item ("magic hate ball",    ({
  "long", "The magic hate ball gives wisdom when you shake it!",
  "shake", (: do_hate :),
  "consult", (: do_consult :),
}));
```

We'll need a function prototype at the top of the code:

```
int do_hate();
int do_consult();
```

And folding in the code we just discussed:

```
int do_consult() {
  string key, value;
  foreach (key,value in previous_responses) {
    tell_object (this_player(), key + " was told '" + value + "'\n");
  }
  return 1;
}
```

Now we can consult to see what everyone has been told by the hate ball:

```
> consult ball
Drakkos was told 'Shut up and die.'   You consult the magic hate ball.
```

Alas though, if no-one has been told anything, we get a fairly empty message:

```
> consult ball
You consult the magic hate ball.
```

We should put a check to ensure that if the mapping is empty, we get the appropriate message. Luckily sizeof works for mappings as well – it gives us the number of keys in a particular mapping, so we can fix that quite easily:

```
int do_consult() {
  string key, value;
  if (sizeof (previous_responses) == 0) {
    tell_object (this_player(), "No-one was told anything.  Now curl up "
      "and die.\n");
  }
  else {
    foreach (key,value in previous_responses) {
      tell_object (this_player(), key + " was told '" + value + "'\n");
    }
  }
  return 1;
}
```

Now we get a much better experience when we try to consult an empty hate ball:

```
> consult ball
No-one was told anything.  Now curl up and die.
You consult the magic hate ball.
```

Thanks for the update, magic hate ball!

Conclusion

It's unlikely you're going to need to do much with mappings at this stage of your Creator Career – they are usually a part of more complicated functionality. It's important though that you understand what they are and how they are manipulated – you'll likely encounter them quite a lot as you read through example code. They are quite unspeakably useful – so useful in fact that when working in an environment where you don't have easy access to them, you miss them with a primal yearning. Luckily that's not a problem for us on Discworld, for we are well supported in our Mapping Use Requirements!

So Long

Introduction

You've done very well to get this far – programming is not an easy task regardless of what anyone tells you. Perhaps the most frustrating thing that goes with learning how to do it is how easily some people 'get it'. It's unfortunate that not everyone finds it as easy as other people, but the willingness to persevere can trump that initial natural understanding. Having the personality to continue when it gets hard is what separates a programmer from a talented amateur.

You may very well fall into the category of talented amateur yourself, in which case this should be a warning for you too – everyone struggles at one point or another to get something working, and if it's been easy all the time then you need to be willing to dig deep and find the will to go on when things start to become difficult.

Anyway, in this brief chapter we shall wrap up our initial foray into LPC and talk about what happens next.

The Past

Learnville isn't exactly what you'd call a finished area – there are bits that are not described, and other bits that have no interesting functionality. That's okay because it's not for the eyes of players – it's a project so that you can see the process by which an area can be built. There are bits of Learnville that are quite sophisticated, and if you've managed to keep up with the code then you can make use of variations of what we've talked about to great effect.

Learning how to develop on Discworld is a two-pronged problem. To begin with, you need to know how programming code works, and this is in itself an entire textbook of knowledge. This is a transferable skill – the coding structures that you've learned as part of this introduction to LPC are transferable to most programming languages. Individuals interested in seeing how transferable their skills actually are invited to explore my Java textbooks at Monkeys At Keyboards.

Combined with the understanding of generic coding structures, you need to learn the specific code that is unique to Discworld – how to create rooms, NPCs and items. Then you need to learn how to link them up, and make them do interesting things.

A good creator is not just a coder – a good creator has an unusual blend of skills. A good creator can architect projects, write well, and produce robust code. A beginner can only be expected to master one of these to start with, but you need to be able to do all three before you can think of yourself as a well-rounded creator. It's not an easy task.

So, let's recap what you've learned so far, and provide some context... there's been a lot of content, and so you may be forgiven for having not really realized how much you've actually done.

In terms of generic programming structures, you've learned:

- Variables and variable scope
- If and Else-if statements
- Switch statements
- Functions and function scope
- For and Foreach loops
- While loops
- Arrays
- Mappings
- Header files

A substantial subset of these are present in most modern programming languages, and understanding how these work in LPC is the first step to understanding how they work in other environments. The syntax may be different, but the concepts are identical.

In terms of what you've learned about Discworld development, we can add to that:

- Inside and outside rooms
- Day and night items, chats and long descriptions
- How to use the linker
- NPCs
- How to use the Armoury
- Searching items
- How to use the taskmaster
- Virtual objects

- •Modifying exits
- •Item and general shops
- •NPC combat actions
- •Adding spells to NPCs
- •Object matching with match_objects_for_existence

That's quite a list of accomplishments, and a well deserved pat on the back is deserved for you for having learned all of these. I won't say mastered, because there's always more to learn about all of these things, and only diligent exploration of the MUD will reveal the more obscure features of all the things we've discussed.

The Present

So, what happens now you've learned all of this? Well, you've undoubtedly been assigned a newbie project by your domain administration, and your focus should be on developing that to as high a standard as you can. There are also small projects available for those who want to apply their new found skills to a small, open project with the intention of it being slotted into an existing area of the game to add richness.

But more than simply your current projects, you should be exploring the codebase. Certainly within your own domain, think of things you enjoyed while playing and hunt out the code that handled it. Read the code, and try to understand how it works. All coding is, to a greater or lesser degree, plagiarism – you can gain huge amounts of understanding by just reading what has come before.

The problem with this of course is that you don't necessarily know if the code that you find is 'good code'. We've had many hundreds of creators of varying degrees of ability over the long years of Discworld, and not all of the code is something you should be emulating. If you wonder whether code is worth using as a template, then feel free to contact me or one of the domain Mentats who will be able to give you guidance.

As far as exploring code outside your domain goes, you should definitely acquaint yourself with the code in /obj/ and /std/. For other domains, ask the domain lord if they mind you scrutinizing the code for inspiration. Hardly anyone will mind, but it's polite to ask. Talk to other creators, especially those who may be as new as you. Part of what makes being a creator so much fun is the social environment in which you function, and you should take full advantage of that.

You should also feel free to talk to members of the Learning domain, particularly the leader of the domain – it's only by talking to people like you that we know what things you want to know about! Asking for information on a particular topic is as helpful to us as it (hopefully) is for you!

The Future

There will come a point, hopefully not too far in the future, when you feel you're ready to learn new things. We've only scratched the surface of the kind of things that get written on Discworld, and there is exciting territory to come. From here, you can move on to Being A Better Creator, Working With Others, and LPC For Dummies 2.

LPC for Dummies one is the first of four related texts. Two of these focus on the softer-side of Discworld creators, and the last is a more advanced guide to LPC development. You should familiarise yourself with all of them - there's a lot discussed across these texts You can think of them as your very own Discworld Creator Manual.

It would be a good idea to at least finish your first domain project before you worry about expanding into this territory – getting a feel for the complexity of development and the things that you wish you knew how to do will make the lessons in later versions of the texts more valuable.

Conclusion

That's us for now! You're done with LPC for Dummies and you can emerge from the room in which you keep your computer... the bright sunlight may hurt at first, but that'll go away with time. We'll see you again soon though, oh yes. We'll see you soon...

Reader Exercises

Introduction

Now that we've reached the end of your introductory voyage through LPC, let's give you some homework to see how well you've learned your lessons. Full answers to (almost) all of these exercises are available, but please attempt them yourself first – you learn more by trying than by reading what other people have done for you.

Exercises

Fill In The Blanks

We haven't done a lot of descriptive work for Learnville, focusing instead on the code we needed to build it. You'll learn a lot however from going over each of the rooms and filling in the blanks. Ensure each room has the following:

1. A day and night long description
2. Day and night add_items
3. Day and night room chats

Importantly, make sure every noun in your long description and in your add_items is included. This is something people will look for in your rooms - believe me.

There is no worked example for this, but if you go through this process and would like to volunteer it as an example for other people, then send it to me and we can make it so.

Connect To Your Workroom

Having gone to the trouble of making this village, you should be able to admire it whenever you like – it should connect into your workroom. However, you're also at some point hopefully going to progress onto Betterville. Create a new room in your directory and call it access_point. Link this room to your workroom, and link Learnville to it too. Add an exit function to the Learnville exit that makes sure only creators can enter your development.

Playing Dress-Up

Most of our NPCs are scantily clad at best. You should improve this situation by making sure they are appropriately dressed. Each NPC should have a shirt, a pair of trousers, underwear, socks, and a pair of shoes or boots. Make sure this is true of all of your NPCs, and that you pick sensible instance of each as to befit their status.

The Very Model of a Modern Major General

Captain Beefy has no weapon, which is somewhat out of theme for a man of his military background. Using the same kind of system as you did for his ring and his boots, create a beefy_sword. It will be saved as a .wep file, and you should look at existing examples of virtual weapons to ensure you set all the right values.

Once you've done this, give him a suitable scabbard (.sca), and make sure both of these are provided to him when he is setup. His sword should be cloned inside the scabbard, and he should be wearing the scabbard along with the rest of his outfit.

Responsiveness

While our NPCs have several responses built into them, it would be nice if they were more responsive generally. Add in code to each of them that makes them respond positively to being greeted, both as a say and as a soul.

Additionally, make them respond to your name with an appropriate amount of fear and cowering awe.

Make them all respond to the names of each other, with a little comment about who they are (those NPCs that already have such responses can be left alone). For example, if I say 'beefy' they could say 'Oh, Beefy – he was our first NPC!'.

Finally, when I ask where certain NPCs are, such as 'Where is Beefy?', they should tell me where I am likely to find them.

Leashing

The code we have for loading NPCs has the impact of 'leashing' them to a specific room – if Beefy is found elsewhere in the village when after_reset is called, he'll be magically transported somewhere else. Change this so that he (and all other relevant NPCs) get moved into the room only if they have no environment. To make sure that they cannot then wander outside Learnville, put a creator check on your exit in access_point

Searching

Add in a mention of a ‘pit of sand’ somewhere in the development, and incorporate a new search function. This search function should allow the player to search five times in a reset period – the first time gives a sword, the second a shield, the third a helm, the fourth a breastplate, and the fifth a pair of blue satin panties. The state of the pit should be shared amongst players, so that if another player comes in and searches, they pick up from where the last one left off.

A Mountain Path

One of the things that changing the move messages with modify_exit allows you to do is create a sense of context to moving. Change the path leading to the village square so that it simulates climbing up a mountain trail. For example

```
You climb the winding trail to the northeast
```

Similarly, when heading back down the trail to access_point, have that described to the player also.

An Open And Shut Case

Making use of the query_fighting lfun, make sure that our shop is not considered to be open when its proprietor is currently engaged in combat.

Mad, Bad and Dangerous to Know

Modify our crazy Stabby so that he cools down over time after having been riled to anger.

Additionally, make it so that he gives a set of tiered warnings to players based on how angry he actually is.

Send Suggestions

Do you have ideas for exercises that would be cool, or are you trying to do something new in Learnville and just can’t make it work? Send them to me, Drakkos, and we can look at incorporating them into this section!