CHAPTER 14

Implementing Scripts

When creating projects as large as role-playing games, you will find it difficult (and foolhardy) to program game-related information in your source code. Your best course is to use external sources (that resemble programming code) called *scripts* for gaming data such as dialogue. In this way, you can control the flow of your game and save time because you don't have to recompile the project every time you make a change. In this chapter, you learn how to create and use a basic scripting system.

In this chapter, you do the following:

- Learn about scripts
- Create your own scripting system
- Use the scripting system
- Apply scripts to your game

Understanding Scripts

When creating a game, you use scripts in much the same way that movie producers use scripts —to control every aspect of your "production." Game scripts are similar to the program code you write when creating your game, except that game scripts are *external* to the gaming engine. Because they are external, you can make quick changes to a script without having to recompile the entire game engine. Imagine having a project with more than one million lines of code and having to recompile the entire project just to change a single line of dialogue!

Scripts are not really difficult to work with, and just about every aspect of your game can benefit from the use of scripts. You can use scripts when navigating menus, controlling combat, handling a player's inventory, and so much more. For example, when developing a game, imagine that you want to present users in combat with a list of magic spells that they regularly use for attack. Say that over the course of developing the game, you decide to change some of those spells. If that spell information is hard-coded, you have a major problem; you must change every instance of the program code that controls the spell, not to mention having to debug and test that code until it's perfect. Why devote so much time on changes such as this one?

Instead, you can write the code for magic spells and their respective effects on the game denizens in several small scripts. Whenever combat commences, these scripts are loaded and the selection of magic spells shown. Once a magic spell is cast, a script processes the effects—from the damage done to the movement and animation of the spell's graphics.

For this book, I was torn between using two different types of scripting systems. One script system involves the use of a language much like C++. You type commands into a script file, compile the file, and execute the compiled script file from within your game. The second script system is an extremely simplified version of the first. Rather than allowing you to type the commands into a file, the system enables you to create scripts by selecting the commands from a predetermined set of commands.

Because I want to get you up and running with your scripting engine as quickly as possible, I opted to use the second script system. This system, which I call the *Mad Lib Scripting* system, works by using a set of predetermined commands, called *actions*, each of which has an associated game function. Take, for example, the actions in Table 14.1—each action has a specific function to perform.

With such a limited set of actions, you really don't need the power of complex compiled script languages; instead, you need the ability to tell the script system which action to use and what options the action should use to perform the gaming function. The great thing about this method is that instead of spouting out lines of code to specify a simple action, you reference the action and options by number.

For example, say that the `Play Sound` action is considered action number four, and the action requires only one entry, the sound number to play. There are only two

Table 14.1 Example Command Actions

Action	Function
Print	Prints a line of text to the screen.
End	Ends script processing.
Move Character	Moves the specified character in a specific direction.
Play Sound	Plays a specific sound effect.

values to store in the script: one number for the action and one number that represents the sound. Using values to represent actions (instead of text) makes processing these types of scripts quick and easy.

Creating a Mad Lib Script System

As I mentioned in the preceding section, I refer to my recommended scripting system as the Mad Lib Script system (or MLS for short) because it closely resembles the old pen-and-paper game of the same name. In Mad Libs (which is founded on the perfect concept for a basic scripting system), you receive a story that is missing numerous words, and your job is to fill in the blanks with hilarious text. While your game's actions represent something other than funny quotes, the idea is perfect for your needs.

In this section, I introduce the concepts of creating a Mad Lib Script system, from developing the actions you use in your scripts to creating a script system (complete with a script editor) that you can insert into your game project.

Designing the Mad Lib Script System

Implementing your own MLS system is easy enough; just create the actions that you want in your game, complete with the blank spots (called *entries*) that need to be filled in by the person creating or editing the scripts. For each action, be sure to provide a list of choices for filling in the blank entries, which can vary in type from a line of text to a numerical value.

You number the actions and the blank entries so that the scripting system can reference them, as illustrated in the following example lists of actions:

1. Character (**NAME**) takes (**NUMBER**) damage.
2. Print (**TEXT**).
3. Play sound effect titled (**SOUND_NAME**).
4. Play music titled (**MUSIC_NAME**).
5. Create object (**OBJECT_NAME**) at coordinates (**XPOS**),(**YPOS**).
6. End script processing.

Each of the six actions has either zero or more blank entries enclosed within parentheses. Each of the blank entries holds either a text string or a number. This list of actions and possible entries (with the type of entry) is called an *action template* (see Figure 14.1 for an example).

Once action templates are in use, you can refer to actions by their numbers rather than by the actions' text (which exists only to make it easier for users to understand which function each action performs). For example, from now on, I can say that I want to implement action #4 using *title.mid* in the first blank entry. When you execute the script, the script system will see the number 4 (action #4) and know that it has only one entry—the filename of the song that you want to load and play.

I trust that you are beginning to see the ease with which you can use this system. Now, I will forgo any more theory so that you can jump right into programming your own MLS system.

NOTE

The MLS scripting system will work for 90 percent of your game. For example, take a look at the PlayStation console in the game *RPG Maker* (by Agetec, Inc.). In *RPG Maker*, you can create your own role-playing games, working off an MLS-type system such as the one I just described; believe me, you can create complex scripts in this game.

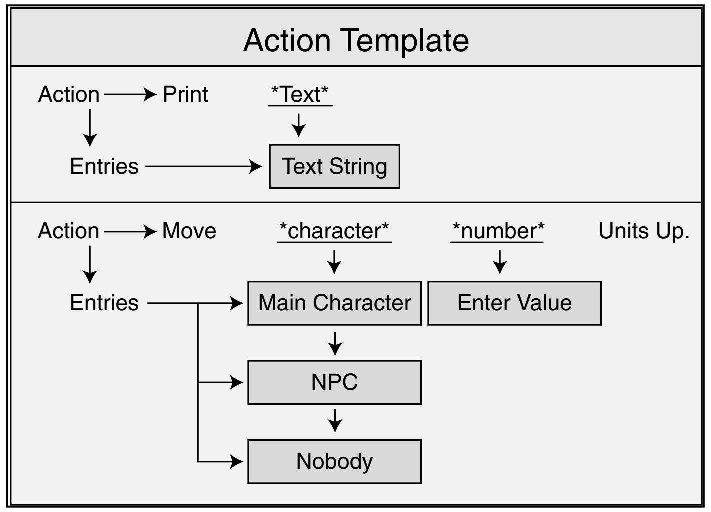


Figure 14.1

An action template is divided into multiple actions, which in turn are split into entries.

Programming the Mad Lib Script System

In order to make your MLS system as powerful as possible, you need to design it so that it supports multiple action templates, with each action template containing an unlimited number of actions. In this way, you can reuse the system for just about any project your heart desires.

To make writing the scripts easier, utilize a script editor program (such as the one you see in the later section, "Working with the MLS Editor") with an action template that enables you to quickly piece together actions and change the blank entries for each action. When a script is complete, you can read the script file into your engine and process each individual action, using the specific entries for each action that was entered via the script editor.

The first order of business is to work with the action templates.

Working with Action Templates

An action template needs to contain a list of actions, complete with text, number of entries, and each entry's data. Recall that each action is numbered by its index within a list, with each blank entry in each action numbered as well. You assign each entry a type (text, integer number, float number, Boolean value, or multiple choice). You also number types, as follows:

0. No entry type
1. Text entry
2. Boolean value
3. Integer number
4. Float number
5. Multiple choice (a choice from a list of text selections)

Each entry type has unique characteristics; strings can be of variable size, numbers can be between any range of two numbers, and Boolean values can either be `TRUE` or `FALSE`. As for multiple choices, each choice has its own text string (the scripts are given a choice from a list, and the index number of the selected choice is used rather than the text).

A sample action might then take this form:

```
Action #1: Spell targets (*MULTIPLE_CHOICE*).
```

```
Possible choices for blank entry #1:
  1. Player character
  2. Spell caster
  3. Spell target
  4. Nobody
```

Imagine that you are using the preceding action and instructing it to use choice #3 as the target. You instruct the script engine to use action #1 with choice #3 for the first blank spot (which is a multiple-choice entry). Using numbers to represent the actions and entries means that the script processor doesn't have to deal directly with code text, which makes processing the scripts easier.

To contain the actions and entries, I've come up with the following structures, which are heavily commented so that you can follow along:

```
// Type of entries (for blank entries)
enum Types { _NONE = 0, _TEXT, _BOOL, _INT, _FLOAT, _CHOICE };

// Structure to store information about a single blank entry
typedef struct sEntry {
  long     Type;        // Type of blank entry (_TEXT, etc.)

  // The following two unions contain the various
  // information about a single blank entry, from
  // the min/max values (for int and float values),
  // and the number of choices in a multiple choice entry.
  //  Text and Boolean entries do not need such info.
  union {
    long   NumChoices;  // # of choices in list
    long   lMin;        // long min. value
    float  fMin;        // float min. value
  };
  union {
    long   lMax;        // long max. value
    float  fMax;        // float max. value
    char **Choices;     // text array for each choice
  };

  // Structure constructor to clear to default values
  sEntry()
  {
    Type       = _NONE;
    NumChoices = 0;
```

```
      Choices    = NULL;
    }

    // Structure destructor to clean up used resources
    ~sEntry()
    {
      // Special case for choice types
      if(Type == _CHOICE) {
        if(NumChoices) {
          for(long i=0;i<NumChoices;i++)
            delete [] Choices[i];  // Delete choice text
        }
        delete [] Choices; // Delete choice array
      }
    }
} sEntry;

// Structure that stores a single action and contains
// a pointer for using linked lists.
typedef struct sAction {
    long      ID;           // Action ID (0 to # actions-1)
    char      Text[256];   // Action text
    short     NumEntries;  // # of entries in action
    sEntry   *Entries;     // Array of entry structures
    sAction  *Next;        // Next action in linked list

    sAction()
    {
      ID         = 0;       // Set all data to defaults
      Text[0]    = 0;
      NumEntries = 0;
      Entries    = NULL;
      Next       = NULL;
    }

    ~sAction()
    {
      delete [] Entries;   // Free entries array
      delete Next;         // Delete next in list
    }
} sAction;
```

You use the two preceding structures, `sEntry` and `sAction`, in conjunction to store the action text as well as the type of each entry. For entries, you select from the enumerated list type (as described earlier in this section). The `sEntry` structure also holds the rules for each entry type (using the two unions).

Because text entries are only buffers of characters, you have no rules to follow for using text entry types. The same goes for Boolean values because they can be only `TRUE` or `FALSE`. Integer and float values need a minimum and maximum range of acceptable values (hence, the min/max variables). There are a number of multiple choices and an array of char buffers that holds the text for each choice.

`sAction` holds the action ID (the action number from the list of actions), the action text, and an array of entries to use for the action. To determine the number of entries in the action (as well as each type), you need to encrypt the action text a bit. To insert an action into the action text, use a tilde (~) character, as shown here:

```
Player ~ gains ~ hit points
```

The two tildes represent two entries. More information is needed about each entry, but how do you obtain information from only two tilde characters? You can't, so you must access the storage format of the action templates to determine what additional information is required for each action.

Action templates are stored as text files, with each action's text enclosed within quotes. Each action that contains entries (marked as tildes in the text) is followed by a list of entry data. Each entry begins with a word that describes the type of entry (`TEXT`, `BOOL`, `INT`, `FLOAT`, or `CHOICE`). Depending on the entry type, further information might follow.

No more information is needed for `TEXT`. The same goes for `BOOL` types. As for `INT` and `FLOAT`, a minimum value and a maximum value are required. At last, the `CHOICE` entry is followed by the number of choices to select from and then by each choice's text (enclosed in quotes).

After you define each entry, you can go on to the next action text. The following example action template file demonstrates each entry type:

```
"Print ~"
  TEXT

"Move character to ~, ~, ~"
  FLOAT 0.0 2048.0
  FLOAT 0.0 2048.0
  FLOAT 0.0 2048.0
```

```
"Character ~ ~ ~ ~ points"
  CHOICE 3
    "Main Character"
    "Caster"
    "Target"
  CHOICE 2
    "Gains"
    "Losses"
  INT 0 128
  CHOICE 2
    "Hit"
    "Magic"

"Set variable ~ to ~"
  INT 0 65535
  BOOL

"End Script"
```

Because the action template doesn't allow comments, I'll explain the actions and entries. The first action (`Print ~`) prints a single text string (using the first entry in the action, entry `0`). The second action takes three float values, each ranging from 0 to 2,048. The third action gives three multiple-choice options as well as an integer value that can range from 0 to 128. Action four demonstrates integer values again, as well as a single Boolean value. Last is action five, which takes no entries.

Loading the action template is a matter of processing a text file and setting up the appropriate structures, which consists of doing string comparisons on words loaded and storing text lines within quotes. This is really an easy process, and in the section, "Putting Together the cActionTemplate Class," you find out exactly how it is done.

The next step is to use the action templates in conjunction with another structure that stores the entry data (which text to display, what number or choice was selected, and so on), which is the purpose of the script entries.

Creating Script Entries

Because the `sEntry` structure contains only the template (guidelines) of the actions and entries, you need another array of structures to store the data for each entry. These new structures include what text to use in a text entry, which Boolean value to use, and which multiple-choice selection to use. This new structure that contains an entry's data is `sScriptEntry`, and is defined as follows:

```
typedef struct sScriptEntry
{
  long   Type;          // Type of entry (_TEXT, _BOOL, etc.)

  union {
    long   IOValue;    // Used for saving/loading
    long   Length;     // Length of text (w/ 0 terminator)
    long   Selection;  // Selection in choice
    BOOL   bValue;     // BOOL value
    long   lValue;     // long value
    float  fValue;     // float value
  };
  char    *Text;        // Text buffer

  sScriptEntry()
  {
    Type = _NONE;  // Clear to default values
    IOValue = 0;
    Text = NULL;
  }

  ~sScriptEntry() { delete [] Text; }  // Delete text buffer
} sScriptEntry;
```

Much like `sEntry`, the `sScriptEntry` holds the actual values to use for each blank entry in the action. Here, you see `Type` again. It describes the type of entry (`_TEXT`, `_BOOL`, and so on). The single union of variables is where the good stuff is, including one variable for the length of the text, one for the multiple choice selection, and one for the integer and float values and the Boolean value.

Take note of two things about `sScriptEntry`. First, a character pointer is outside the union (because both `Length` and `Text` are used to store text data); second, an additional variable called `IOValue` is included in the union. You use `IOValue` to access the union variables to save and load the entry data.

To demonstrate how to store each action's entry data into an `sScriptEntry` structure (or structures if there is more than one entry), review the following action:

```
"~ player's health by ~"
  CHOICE 2
    "Increase"
    "Decrease"
  INT 0 65535
```

Depending on multiple choice selection, the preceding action either increases or decreases the player's health by a set amount ranging from 0 to 65535. Because there are two entries (a multiple choice and an integer), you need two sScriptEntry structures:

```
sScriptEntry Entries;

// Configure multiple choice - set to first choice
Entries[0].Type = _CHOICE;
Entries[0].Selection = 0; // Increase

// Configure integer - set to 128
Entries[1].Type = _INT;
Entries[1].lValue = 128;
```

When dealing with the script entries, the most difficult part crops up when many entries are in a complete script. Each action in the script requires a matching sEntry structure, which in turn might contain a number of sScriptEntry structures. Before you know it, you can become knee-deep in structures—talk about a mess! To better handle a script's structures, you need another structure that tracks each entry that belongs to the script actions:

```
typedef struct sScript
{
  long          Type;        // 0 to (number of actions-1)
  long          NumEntries;  // # entries in this script action
  sScriptEntry *Entries;     // Array of entries

  sScript      *Prev;        // Prev in linked list
  sScript      *Next;        // Next in linked list

  sScript()
  {
    Type = 0;          // Clear to defaults
    NumEntries = 0;
    Entries = NULL;
    Prev = Next = NULL;
  }

  ~sScript()
  {
    delete [] Entries; // Delete entry array
    delete Next;       // Delete next in linked list
  }
} sScript;
```

You use the sScript structure to contain a single action, as well as maintain a linked list of further sScript structures that constitutes an entire script. The Type variable can range from zero to the number of actions in the action template minus one. If you have ten actions in the action template, Type can range from zero to nine.

To make processing easier, store the number of entries in the NumEntries variable. The value in NumEntries must match the number-of-entries variable in the action template. From there, allocate an array of sScriptEntry structures to store the data for each entry in the action template. If two entries are in the associated action, you need to allocate two sScriptEntry structures.

Lastly, there are the two pointer variables, Prev and Next, in sScript. These two pointers maintain a linked list of the entire script. To construct a linked list of sScript structures (much as illustrated in Figure 14.2), start with a root structure that represents the first action in the script. You then link sScript structures via the Next and Prev variables, as shown here:

```
sScript *ScriptRoot = new sScript();
sScript *ScriptPtr = new sScript;
ScriptRoot->Next = ScriptPtr;  // Point to second action
ScriptPtr->Prev = ScriptRoot;  // Point back to root
```

At this point, you can start at the root of the script and traverse down the entire script with the following code:

```
void TraverseScript(sScript *pScript)
{
  while(pScript != NULL) { // loop until no more script actions
    // Do something with pScript
    // pScript->Type holds the script action ID
    pScript = pScript->Next; // Go to next script action
  }
}
```

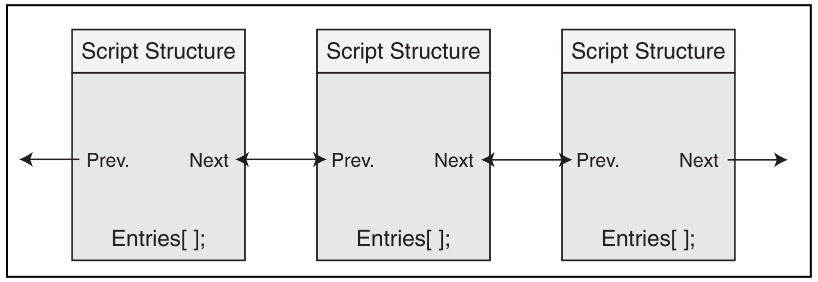


Figure 14.2
A script-action linked list uses Prev *and* Next *variables to link the entire script. Each script action has it own array of script entries.*

You can also quickly load and save scripts by using linked lists, as illustrated in the following two functions:

```
BOOL SaveScript(char *Filename, sScript *ScriptRoot)
{
  FILE *fp;
  long i, j, NumActions;
  char Text[256];
  sScript *ScriptPtr;

  // Make sure there's some script actions
  if((ScriptPtr = ScriptRoot) == NULL)
    return FALSE;

  // Count the number of actions
  NumActions = 0;
  while(ScriptPtr != NULL) {
    NumActions++;                 // Increase count
    ScriptPtr = ScriptPtr->Next; // Next action
  }

  // Open the file for output
  if((fp=fopen(Filename, "wb"))==NULL)
    return FALSE;  // return a failure

  // Output # of script actions
  fwrite(&NumActions, 1, sizeof(long), fp);

  // Loop through each script action
  ScriptPtr = ScriptRoot;
  for(i=0;i<NumActions;i++) {

    // Output type of action and # of entries
    fwrite(&ScriptPtr->Type, 1, sizeof(long), fp);
    fwrite(&ScriptPtr->NumEntries, 1, sizeof(long), fp);

    // Output entry data (if any)
    if(ScriptPtr->NumEntries) {
      for(j=0;j<ScriptPtr->NumEntries;j++) {

        // Write entry type and data
```

```
        fwrite(&ScriptPtr->Entries[j].Type, 1,sizeof(long), fp);
        fwrite(&ScriptPtr->Entries[j].IOValue,1,sizeof(long),fp);

         // Write text entry (if any)
         if(ScriptPtr->Entries[j].Type == _TEXT &&              \
            ScriptPtr->Entries[j].Text != NULL)
           fwrite(ScriptPtr->Entries[j].Text, 1,                \
                  ScriptPtr->Entries[j].Length, fp);
      }
    }

    // Go to next script structure in linked list
    ScriptPtr = ScriptPtr->Next;
  }

  fclose(fp);

  return TRUE; // return a success!
}

sScript *LoadScript(char *Filename, long *NumActions)
{
  FILE *fp;
  long i, j, Num;
  char Text[2048];
  sScript *ScriptRoot, *Script, *ScriptPtr = NULL;

  // Open the file for input
  if((fp=fopen(Filename, "rb"))==NULL)
    return NULL;

  // Get # of script actions from file
  fread(&Num, 1, sizeof(long), fp);

  // Store number of actions in user supplied variable
  if(NumActions != NULL) *NumActions = Num;

  // Loop through each script action
  for(i=0;i<Num;i++) {

    // Allocate a script structure and link in
```

```
      Script = new sScript();
      if(ScriptPtr == NULL)
        ScriptRoot = Script;  // Assign root
      else
        ScriptPtr->Next = Script;
      ScriptPtr = Script;

      // Get type of action and # of entries
      fread(&Script->Type, 1, sizeof(long), fp);
      fread(&Script->NumEntries, 1, sizeof(long), fp);

      // Get entry data (if any)
      if(Script->NumEntries) {

        // Allocate entry array
        Script->Entries = new sScriptEntry[Script->NumEntries]();

        // Load in each entry
        for(j=0;j<Script->NumEntries;j++) {

          // Get entry type and data
          fread(&Script->Entries[j].Type, 1, sizeof(long), fp);
          fread(&Script->Entries[j].IOValue, 1, sizeof(long), fp);

          // Get text (if any)
          if(Script->Entries[j].Type == _TEXT &&                 \
             Script->Entries[j].Length) {
            // Allocate a buffer and get string
            Script->Entries[j].Text =                            \
                    new char[Script->Entries[j].Length];
            fread(Script->Entries[j].Text, 1,                    \
                  Script->Entries[j].Length, fp);
          }
        }
      }
    }

    fclose(fp);

    return ScriptRoot;
  }
```

Given the root script structure in a linked list, `SaveScript` will output each script structure's data, which includes the action number, the number of entries to follow, the entry data, and the optional text of a text entry. The entire linked list of `sScript` structure is written to the file.

The `LoadScript` function opens the script file in question and builds a linked list of `sScript` structures from the data it loads. `sScriptEntry` structures are allocated on-the-fly, as well as the `sScript` structures that construct the linked list. When complete, the `LoadFile` function sets `NumActions` to the number of script actions loaded and returns a pointer to the root script structure.

Putting Together the cActionTemplate Class

You now understand the structure used for action templates and for containing the script data. Now, it's time to put them all together in order to create a working class that loads and processes scripts:

```
class cActionTemplate {
  private:
    long     m_NumActions;    // # of actions in template
    sAction *m_ActionParent;  // list of template actions

    // Functions for reading text (mainly used in actions)
    BOOL GetNextQuotedLine(char *Data, FILE *fp, long MaxSize);
    BOOL GetNextWord(char *Data, FILE *fp, long MaxSize);

  public:
    cActionTemplate();
    ~cActionTemplate();

    // Load and free an action template
    BOOL     Load(char *Filename);
    BOOL     Free();

    // Get # actions in template, action parent,
    // and specific action structure.
    long     GetNumActions();
    sAction *GetActionParent();
    sAction *GetAction(long Num);

    // Get a specific type of sScript structure
    sScript *CreateScriptAction(long Type);

    // Get info about actions and entries
    long     GetNumEntries(long ActionNum);
    sEntry  *GetEntry(long ActionNum, long EntryNum);

    // Expand action text using min/first/TRUE choice values
    BOOL     ExpandDefaultActionText(char *Buffer, sAction *Action);

    // Expand action text using selections
    BOOL     ExpandActionText(char *Buffer, sScript *Script);
};
```

The only functions in this code that you haven't seen in this chapter are GetNextQuotedLine and GetNextWord. The GetNextQuotedLine function scans the file in question for a line of text enclosed within quotes, while the GetNextWord function reads in the next text word from a file. Both functions take a pointer to a data buffer in which to store the text, the file access pointer, and the maximum size of the data buffer (to avoid overflow):

```
BOOL cActionTemplate::GetNextQuotedLine(char *Data,           \
                                        FILE *fp, long MaxSize)
{
  int c;
  long Pos = 0;

  // Read until a quote is reached (or EOF)
  while(1) {
    if((c = fgetc(fp)) == EOF)
      return FALSE;

    if(c == '"') {
      // Read until next quote (or EOF)
      while(1) {
        if((c = fgetc(fp)) == EOF)
          return FALSE;

        // Return text when 2nd quote found
        if(c == '"') {
          Data[Pos] = 0;
          return TRUE;
        }
        // Add acceptable text to line
        if(c != 0x0a && c != 0x0d) {
          if(Pos < MaxSize-1)
            Data[Pos++] = c;
        }
      }
    }
  }
}

BOOL cActionTemplate::GetNextWord(char *Data, FILE *fp,        \
                                  long MaxSize)
{
  int  c;
  long Pos = 0;

  // Reset word to empty
  Data[0] = 0;

  // Read until an acceptable character found
  while(1) {
    if((c = fgetc(fp)) == EOF) {
      Data[0] = 0;
      return FALSE;
    }

    // Check for start of word
    if(c != 32 && c != 0x0a && c != 0x0d) {
      Data[Pos++] = c;

      // Loop until end of word (or EOF)
      while((c=fgetc(fp)) != EOF) {
        // Break on acceptable word separators
        if(c == 32 || c == 0x0a || c == 0x0d)
          break;

        // Add if enough room left
        if(Pos < MaxSize-1)
          Data[Pos++] = c;
      }
      // Add end of line to text
      Data[Pos] = 0;

      return TRUE;
    }
  }
}
```

Using the `GetNextQuotedLine` and `GetNextWord` functions, you can scan input files for text that describes the actions, which is the purpose of the `cActionTemplate::Load` function:

```
BOOL cActionTemplate::Load(char *Filename)
{
  FILE *fp;
  char Text[2048];
  sAction *Action, *ActionPtr = NULL;
  sEntry  *Entry;
  long i, j;

  // Free previous action structures
  Free();

  // Open the action file
  if((fp=fopen(Filename, "rb"))==NULL)
    return FALSE;

  // Keep looping until end of file found
  while(1) {
    // Get next quoted action
    if(GetNextQuotedLine(Text, fp, 2048) == FALSE)
      break;

    // Quit if no action text
    if(!Text[0])
      break;

    // Allocate an action structure and append it to list
    Action         = new sAction();
    Action->Next   = NULL;
    if(ActionPtr == NULL)
      m_ActionParent = Action;
```

```
    else
      ActionPtr->Next = Action;
    ActionPtr = Action;

    // Copy action text
    strcpy(Action->Text, Text);

    // Store action ID
    Action->ID = m_NumActions;

    // Increase the number of actions loaded
    m_NumActions++;

    // Count the number of entries in the action
    for(i=0;i<(long)strlen(Text);i++) {
      if(Text[i] == '~')
        Action->NumEntries++;
    }

    // Allocate and read in entries (if any)
    if(Action->NumEntries) {
      Action->Entries = new sEntry[Action->NumEntries]();
      for(i=0;i<Action->NumEntries;i++) {
        Entry = &Action->Entries[i];

        // Get type of entry
        GetNextWord(Text, fp, 2048);

        // TEXT type, no data follows
        if(!stricmp(Text, "TEXT")) {
          // Set to text type
          Entry->Type = _TEXT;
        } else

        // INT type, get min and max values
        if(!stricmp(Text, "INT")) {
          // Set to INT type and allocate INT entry
          Entry->Type = _INT;

          // Get min value
          GetNextWord(Text, fp, 2048);
    Entry->lMin = atol(Text);

    // Get max value
    GetNextWord(Text, fp, 2048);
    Entry->lMax = atol(Text);
  } else

  // FLOAT type, get min and max values
  if(!stricmp(Text, "FLOAT")) {
    // Set to FLOAT type and allocate FLOAT entry
    Entry->Type = _FLOAT;

    // Get min value
    GetNextWord(Text, fp, 2048);
    Entry->fMin = (float)atof(Text);

    // Get max value
    GetNextWord(Text, fp, 2048);
    Entry->fMax = (float)atof(Text);
  } else

  // BOOL type, no options
  if(!stricmp(Text, "BOOL")) {
    // Set to BOOL type and allocate BOOL entry
    Entry->Type = _BOOL;
  } else

  // CHOICE type, get number of entries and entry's texts
  if(!stricmp(Text, "CHOICE")) {
    // Set to CHOICE type and allocate CHOICE entry
    Entry->Type = _CHOICE;

    // Get the number of choices
    GetNextWord(Text, fp, 1024);
    Entry->NumChoices = atol(Text);
    Entry->Choices = new char[Entry->NumChoices];

    // Get each entry text
    for(j=0;j<Entry->NumChoices;j++) {
      GetNextQuotedLine(Text, fp, 2048);
      Entry->Choices[j] = new char[strlen(Text)+1];
```

```
            strcpy(Entry->Choices[j], Text);
          }
        }
      }
    }
  }

  fclose(fp);

  return TRUE;
}
```

Using the cActionTemplate::Load function, you can open a text file and begin scanning through it. With the beginning of each iteration, the next line of text enclosed in quotes (an action) is loaded in a new sAction structure and then examined for tilde characters. If tilde characters are found, the remaining information is loaded and parsed. This process continues until the end of the file is found.

Moving on, the next questionable function in cActionTemplate is CreateScriptAction; it takes an action number and returns an initialized sScript structure that is set up to store the number of entries to match the action. You can directly parse the sScript structure from this point on to access data contained within the actions and entries (which is how the MLS editor and samples do it):

```
sScript *cActionTemplate::CreateScriptAction(long Type)
{
  long i;
  sScript *Script;
  sAction *ActionPtr;

  // Make sure it's a valid action - Type is really the
  // action ID (from the list of actions already loaded).
  if(Type >= m_NumActions)
    return NULL;

  // Get pointer to action
  if((ActionPtr = GetAction(Type)) == NULL)
    return NULL;

  // Create new sScript structure
  Script = new sScript();
```

```
    // Set type and number of entries (allocating a list)
    Script->Type       = Type;
    Script->NumEntries = ActionPtr->NumEntries;
    Script->Entries    = new sScriptEntry[Script->NumEntries]();

    // Set up each entry
    for(i=0;i<Script->NumEntries;i++) {
      // Save type
      Script->Entries[i].Type = ActionPtr->Entries[i].Type;

      // Set up entry data based on type
      switch(Script->Entries[i].Type) {
        case _TEXT:
          Script->Entries[i].Text = NULL;
          break;

        case _INT:
          Script->Entries[i].lValue = ActionPtr->Entries[i].lMin;
          break;

        case _FLOAT:
          Script->Entries[i].fValue = ActionPtr->Entries[i].fMin;
          break;

        case _BOOL:
          Script->Entries[i].bValue = TRUE;
          break;

        case _CHOICE:
          Script->Entries[i].Selection = 0;
          break;
      }
    }

    return Script;
  }
```

NOTE

I didn't include the script saving or loading functions because they are not part of the action templates. However, you can modify the saving and loading functions for each application as you see fit. This is also the case for this chapter's two sample programs, MlsEdit and MlsDemo, which you can find on the CD-ROM at the back of this book (both programs are in the \BookCode\Chap14 directory).

Last in `cActionTemplate` are the two final functions: `ExpandDefaultActionText` and `ExpandActionText`. Both functions take the action text and replace the tilde characters inside with more understandable text, such as an integer number or the selected multiple-choice text. The difference between the functions is that `ExpandDefaultActionText` expands text with any entry data; it simply picks the minimum values or first multiple-choice entry. `ExpandActionText`, expands the action text using the data contained in the supplied `sScript` structure. Both functions are used only in the script editor to make sense of the data contained with the action template and script structures—you can find the code for them on the CD-ROM (in the MLS Script Editor project).

With an understanding of the action templates and script structures, you can start piecing them together and putting MLS to good use, which all starts with the Mad Lib script editor.

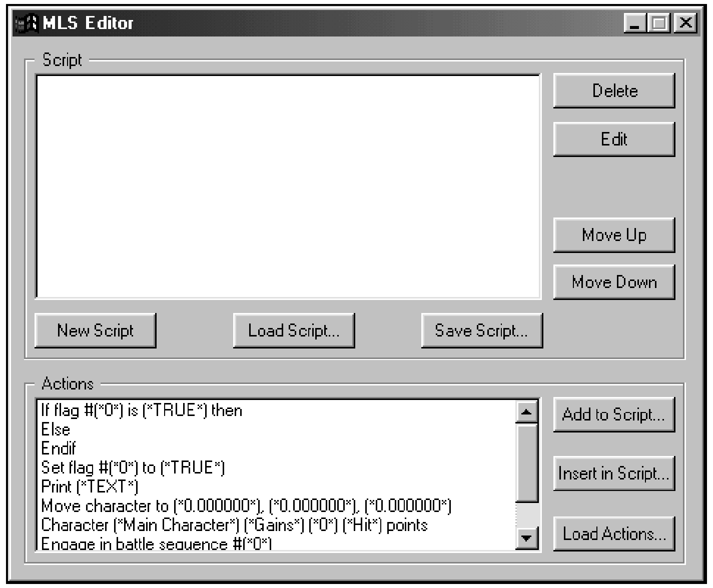
Working with the MLS Editor

An MLS system works only with numbers: the number that represents an action, the number of entries to follow, and numbers to represent the entry data. Computers work well with numbers, but we need more. You need to construct scripts in comprehensible lines of text and let a script editor convert the text you enter into a series of numerical representations that a script system can handle.

During the editing of a script, dealing with numbers is not for us, so the editor also has the job of loading and converting those numbers back into lines of text that is easy for us to read. So, to clear up matters, you only need to construct a script using a series of text commands, and let the script editor and engine convert those commands into their numerical representations and vice versa.

The Mad Lib script editor imports the text that represents the actions and provides the user with the ability to edit a list of actions and modify the blank entry spots with each action. Figure 14.3 shows the MLS editor I created for the book. The script list box, which contains the currently edited script, is at the top of the MLS application window. The actions from the action template are listed at the bottom of the window. The various buttons used to construct the scripts are spread around the window.

You will find using the script editor to be very intuitive. You have options for loading a set of actions, loading and saving a script, creating a new script, and adding, removing, and modifying script entries (as well as for moving their entries up or down the list). The actions used by the editor are stored in action template files.



```
If flag #(*0*) is (*TRUE*) then
Else
Endif
Set flag #(*0*) to (*TRUE*)
Print (*TEXT*)
Move character to (*0.000000*), (*0.000000*), (*0.000000*)
Character (*Main Character*) (*Gains*) (*0*) (*Hit*) points
Engage in battle sequence #(*0*)
```

Figure 14.3

This MLS editor contains all the essentials for creating and editing scripts.

As for the actual script entries, the editor makes use of the `sScript` and `sScriptEntry` structures to store the current script being edited, and are saved and loaded just as you have already seen.

To start your MLS editing session, go ahead and load up an action template or use the default action template, which is titled `default.mla` (you can find it in the \BookCode\Chap14\Data directory). Then you can begin adding, inserting, and editing script entries by using the respective buttons in the editor's application window. Table 14.2 explains what each button does in the script editor.

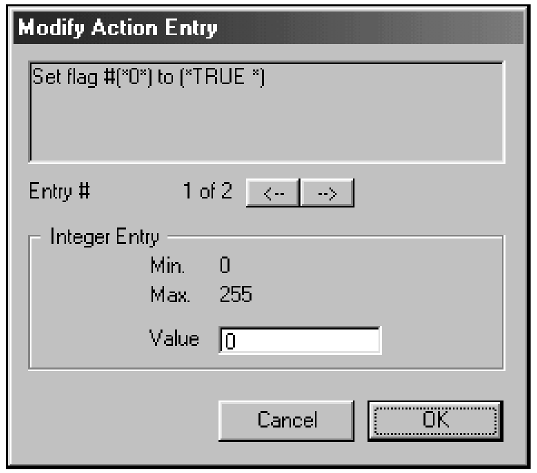
As you begin adding actions to the script (using Add to Script, or Insert in Script), notice that the action text is expanded and added to the script list box (the list box at the top of the script editor). The script actions are stored from the top down, with the root of the script being the topmost script action. Processing of the scripts starts at the top and continues downward, much like typical C/C++ code.

Notice that each time you add, insert, or edit a script entry, the Modify Action Entry dialog box appears (see Figure 14.4). You use this dialog box to modify the script action entries.

In the Modify Action Entry dialog box, you see various controls for modifying the script action entries. The dialog box provides two places to type text. You use the first one

Table 14.2 The MLS Editor Buttons

Button	Function
Delete	Deletes the currently selected line from the script list box.
Edit	Edits the entries from the currently selected line from the script list box.
Move Up	Moves the currently selected script action up in the list box.
Move Down	Moves the currently selected script action down in the list box.
New Script	Removes all script actions from memory and starts with a fresh slate.
Load Script	Loads a script file from disk (files with an .MLS extension).
Save Script	Saves a script file to disk (files with an .MLS extension).
Add to Script	Adds the currently selected action (from the action list) to the end of the script list. This automatically opens the Modify Action Entry dialog box as well.
Insert in Script	Inserts the currently selected action (from the action list) into the selected line in the script list. Also opens the Modify Action Entry dialog box.
Load Actions	Loads a new action template file (files with an extension .MLA). This also forces the current script to be cleared.



Set flag #(*0*) to (*TRUE *)

Entry # 1 of 2

Integer Entry
Min. 0
Max. 255
Value 0

Figure 14.4

Use the Modify Action Entry dialog box to quickly navigate and modify the script's action entries.

(at the top of the dialog box) to type an entry's text or the minimum and maximum ranges for values; in the second one, you type values relevant to the entry. Boolean values have two radio buttons, one to select a `TRUE` value and another to select a `FALSE` value. The dialog box provides a list box for multiple-choice selections.

A few controls that are common to each type of entry are at the top of the Modify Action Entry dialog box. First is the box that displays the action text (with the selected entries expanded in the text). Next is an Entry #, a text box that displays the entry number currently being editing, as well as the number of entries in the action. To navigate the entries, you use two buttons—the previous entry button (represented by an arrow pointing left) and the next entry button (represented by an arrow pointing right). Clicking either button forces the current entry to be updated and the next entry's data to be displayed.

At the bottom of the Modify Action Entry dialog box are two more buttons—OK and Cancel. The Cancel button is displayed only when you add an action. When you select an action to edit from the list, the Cancel button is not shown, which means that all the changes you make to an entry are used whenever OK is clicked, so make sure that you don't modify anything if that's not your intention. Clicking OK accepts all entry data and adds, inserts, or modifies the action selected in the MLS Editor dialog box.

NOTE

The code for the MLS editor is on the CD-ROM at the back of this book (look for \BookCode\Chap14\MLSEdit\).

The script editor comes with a sample action template and script to help you get started. The real power comes when you start constructing your own action templates, tailored for your game project. After you create the action templates and construct your script, you are ready to start using them in your own project.

Executing Mad Lib Scripts

Whew! I can honestly say the hardest part is over, as executing the scripts is child's play at this point. You can now toss the action templates out the door because you work with only the `sScript` and `sScriptEntry` structures from here on out.

The first step to working with a script is to load it into memory, which you accomplish using the `LoadScript` function (refer also to the section "Creating Script Entries" for more on this function):

```
long NumActions;
sScript *LoadedScript = LoadScript("Script.mls", &NumActions);
```

From this point on, your game engine just iterates the script-linked list in order to execute each action. This requires a bit of hard-coding because the actions are known only by numbers at this point (so you must know what each action does). Here's an example that iterates the preceding loaded script and looks for `Print` actions (action `0`), which contain a single entry (the text to print):

```
sScript *ScriptPtr = LoadedScript; // Start at root

// Loop through all script actions in list
while(ScriptPtr != NULL) {
  // Is it an action 0?
  if(ScriptPtr->Type == 0) {
    // This action definitely has one entry, the text.
    // Display the text in a message box
    MessageBox(NULL, ScriptPtr->Entries[0].Text, "TEXT", MB_OK);
  }

  // Go to next action in script
  ScriptPtr = ScriptPtr->Next;
}
```

Although the preceding is nothing more than a few lines of code, it demonstrates the awesome potential of processing the scripts. With a little ingenuity, you could use MLS to handle some major scripting duties.

How about using conditional `if...then...else` statements? You know, those statements that determine whether a condition is true or false and, depending on the outcome, process a different sequence of actions. Take for example the following C code:

```
BOOL GameFlags[256];  // Some game flags defined in the game

if(GameFlags[0] == TRUE) {
  // Print a message and set flag to FALSE
  MessageBox(NULL, "It's TRUE!", "Message", MB_OK);
  GameFlags[0] = FALSE;
} else {
  // Print a message
  MessageBox(NULL, "It's FALSE.", "Message", MB_OK);
}
```

Based on the value contained in the GameFlags array, a different block of code is processed. By creating a few actions and a slight reworking of the script processing code, you could enjoy the benefits of using if...then...else statements in MLS as well. First, check out the action template:

```
"If GameFlag ~ equals ~ then"
  INT 0 255
  BOOL
"Else"
"EndIf"
"Set GameFlag ~ to ~"
  INT 0 255
  BOOL
"Print ~"
  TEXT
```

There is nothing special here because the real work is done in the script-processing code:

```
// pScript = pre-loaded script that contains the following:
//  "If GameFlag (0) equals (TRUE) then"
//    "Print (It's TRUE!)"
//    "Set GameFlag (0) to (FALSE)"
//  "Else"
//    "Print (It's FALSE.)"
//  "EndIf"

// Action processing functions
sScript *Script_IfThen(sScript *Script);
sScript *Script_Else(sScript *Script);
sScript *Script_EndIf(sScript *Script);
sScript *Script_SetFlag(sScript *Script);
sScript *Script_Print(sScript *Script);

// The script action execution structure
typedef struct sScriptProcesses {
  sScript *(*Func)(sScript *ScriptPtr);
} sScriptProcesses;

// List of script action function structures
sScriptProcesses ScriptProcesses[] = {
  { Script_IfThen     },
```

```
  { Script_Else       },
  { Script_EndIf      },
  { Script_SetFlag    },
  { Script_Print      }
};

BOOL GameFlags[256];  // The games flags array

void RunScript(sScript *pScript)
{
  // Clear the GameFlags array to FALSE for this example
  for(short i=0;i<256;i++)
    GameFlags[i] = FALSE;

  // Scan through script and process functions
  while(pScript != NULL) {
    // Call script function and break on NULL return value.
    // Any other return type is the pointer to the next
    // function, which is typically pScript->Next.
    pScript = ScriptProcesses[pScript->Type].Func(pScript);
  }
}

sScript *Script_IfThen(sScript *Script)
{
  BOOL Skipping; // Flag is skipping script actions

  // See if a flag matches second entry
  if(g_Flags[Script->Entries[0].lValue % 256] ==            \
             Script->Entries[1].bValue)
    Skipping = FALSE;
  else
    Skipping = TRUE;

  // At this point, Skipping states if the script actions
  // need to be skipped due to a conditional if..then statement.
  // Actions are further processed if skipped = FALSE, looking
  // for an else to flip the skip mode, or an endif to end
  // the conditional block.

  // Go to next action to process
  Script = Script->Next;

  while(Script != NULL) {
    // if Else, flip skip mode
    if(Script->Type == 1)
      Skipping = (Skipping == TRUE) ? FALSE : TRUE;

    // break on EndIf
    if(Script->Type == 2)
      return Script->Next;

    // Process script function in conditional block
    // making sure to skip actions when condition not met.
    if(Skipping == TRUE)
      Script = Script->Next;
    else {
      if((Script = ScriptProcesses[Script->Type].Func(Script)) == NULL)
        return NULL;
    }
  }
  return NULL; // end of script reached
}

sScript *Script_SetFlag(sScript *Script)
{
  // Set a Boolean flag
  GameFlags[Script->Entries[0].lValue % 256] =                \
            Script->Entries[1].bValue;
}

sScript *Script_Else(sScript *Script) { return Script->Next;  }
sScript *Script_EndIf(sScript *Script) { return Script->Next; }

sScript *Script_Print(sScript *Script)
{
  MessageBox(NULL, Script->Entries[0].Text, "Text", MB_OK);
  return Script->Next;
}
```

You can see that the real magic is in the `Script_IfThen` statement, which is a recursive function that processes all script actions contained within a pair of `if...then` and `EndIf` actions. The `Else` action does a simple job of switching processing modes (from no processing to processing), based on the original value of the `Skipping` variable.

Now that is power, and if you need a little more convincing, I suggest that you check out some later chapters that use the MLS system, such as Chapter 16, “Controlling Players and Characters,” and Chapter 20, “Putting Together a Full Game.” Both chapters demonstrate the use of scripts when interacting with game characters.

Applying Scripts to Games

From the beginning of your project, expect to implement scripts in every game-related detail. For example, scripts come in handy when dealing with dialogue and cinemas all the way down to spell effects and inventory handling. In fact, creating your game engine to accept scripts for the majority of in-game data produces a very open-source and efficient project.

In Chapter 20, you learn just how to apply the scripts to your various game components, such as the combat and inventory system. As for now, you might want to become familiar with the whole script concept by checking out the sample program MlsDemo, which is on this book’s CD-ROM.

Wrapping Up Scripting

The scripting method introduced in this chapter is very powerful when used correctly, and in most cases, will be just the right system for your game project. Advanced readers who want to develop their own “real” script language (one that resembles C++, for example) might want to acquire a good book on compilers, specifically one that utilizes lex and yacc (two programs that process text and grammar). One such book, aptly titled *lex & yacc,* is a great guide to learning the basics on creating a script-parsing language processor. Turn to Appendix C, “Recommended Reading,” for more information on the book.

If you are intrigued by the power behind the MLS system, before beginning your project, you might create a set of action templates that will carry you through the entire game. In this chapter, I discussed some of the simpler techniques for doing so, but I’m sure that you can build on this information and come up with other great uses for MLS.

Programs on the CD-ROM

Two programs that demonstrate the code discussed in this chapter are located on the CD-ROM at the back of this book. You can find the following programs in the \BookCode\Chap14\ directory:

- **MlsEdit.** A Mad Lib Script editor program that is perfect for putting together scripts for your project. Location: \BookCode\Chap14\MlsEdit\.
- **MlsDemo.** A small project that demonstrates the parsing of Mad Lib Scripts created with the MLS editor. Location: \BookCode\Chap14\MlsDemo\.