COM S 327, Fall 2016
Programming Project 1.07

I've combined two previous assignments into one, here. Formerly, students implemented monster parsing one week, then item parsing the next. The two assignments are very similar, to an extent that most students found the second assignment uninstructive. I don't want to bore you with what is essentially a repeated assignment, and by cutting a week out of the old assignments, we can add a week of new functionality at the end.

This assignment specification combines monster and item parsing into one document. Your assignment is to:

- *Rename your C files to C++ files, if necessary modify your makefile, and build the entire program with g++. You will have to make some changes to the sources to accomplish this (mostly casts), but it should mostly "just work".*
- *Implement the "Parsing Monster Definitions" assignment, below.*

You may, optionally, implement the "Parsing Item Definitions" assignment, as well. I will implement both, and for students not using my code, I will release my solution in a standalone form so that you will not have to *implement item parsing if you don't want to or don't have time. You will, however, have to have an item parsing solution, whether you write your own or use mine, as next week's assignment will depend on it.*

Parsing Monster Definitions

We've been randomly generating our NPCs, and while that's not a totally unworkable way to build a real game, it does have some major disadvantages; for instance, it prevents the player from recognizing monsters and knowing, from previous encounters, what to expect from them. As Roguelikes are tactics games, knowing what to expect is essential to success. We've also been very simplistic, with no concept of hitpoints, attack strength, defensive abilities, etc. It's time to increase the complexity of our NPCs, and later we'll work on the PC.

To create more complex NPCs, we could hard code all of the monsters we want. That's as simple as defining an NPC class or structure, creating an array of them, one for each monster, and assigning values to the fields. Problem solved. And if we think of an interesting, new monster that we want to add to the game? We'll need to increase the size of the array, set the fields in the new slot, and recompile. So, essentially, only a developer can do it. A more flexible method will use a text file in which monsters are defined; the game only needs to know how to parse it, and we can change, add, and remove all the monsters we want without ever recompiling.

We're thinking about defining an editable file format to describe NPCs, so, in fact, we're thinking about defining a language and implementing a parser for it. Language design and parsing in the general sense are beyond the scope of this course, but we already know enough to be able to scan tokens from the input with clearly defined delimiters, so we'll design our language so that it is simple enough that we can easily parse it with the tools we have. Students who are interested in implementing more complex monster definition files with more complex parsing requirements should look into *lex*, *yacc*, and *recursive descent*.

What characteristics do we want our monsters to have? So far, they have a symbol, a speed, and perhaps telepathy, intelligence, tunneling ability, and eraticism. These provide a good start. Let's add a color, a name, a description, hitpoints (health), attack damage, and a general "abilities" field, where *intelligence* and *telepathy* are just two possibilities. Given time we'd add other things, including types of attacks: dragons can breathe fire, basilisks can paralyze with a look, and sorcerers can cast spells.

Many of these things are numeric. We could provide a mean and a standard deviation, then roll for a value from a Gaussian. But this is a role-playing game. Role-playing games traditionally use dice, and we're going to stick with tradition on this. We'll specify our values with the format:

<*base*> + <*dice*> d <*sides*>

where <*base*> is a constant offset (maybe zero), <*dice*> is the number of dice to roll (also maybe zero), and <*sides*> is the number of sides on each of those dice. So a numerical specification of `9+3d8` means to roll a number with a minimum value of 12, a maximum value of 33, and an expected value of 22.5. The distribution of rolls for a `9+3d8` looks like this:

Roll	Probability	Cumulative Probability
12	0.2	0.2
13	0.6	0.8
14	1.2	2.0
15	2.0	3.9
16	2.9	6.8
17	4.1	10.9
18	5.5	16.4
19	7.0	23.4
20	8.2	31.6
21	9.0	40.6
22	9.4	50.0
23	9.4	59.4
24	9.0	68.4
25	8.2	76.6
26	7.0	83.6
27	5.5	89.1
28	4.1	93.2
29	2.9	96.1
30	2.0	98.0
31	1.2	99.2
32	0.6	99.8
33	0.2	100.0

I suggest that you store these as three values, for example, *base*, *dice*, and *sides*. They can then be used later to instantiate a dice class object.

Let's define some keywords for all of the parameters of our monsters. In the table below, *numerical* means a descriptor for dice as defined above. If you want an actual constant, then you have zero dice, i.e., `12+0d1`.

Parameter	Keyword	Description
Name	`NAME`	A name to describe the monster, for example, a "slime", an "amazon", a "wyvern", or "Sauron". Terminated by a newline
Description	`DESC`	A textual description of the monster. Beginning on the next line after the keyword, terminated by a period on a line by itself, and limited to a width of 77 characters (the newline must appear at or before byte 78).
Color	`COLOR`	A list of space-delineated color name keywords (see color information below), used to color the monster in the dungeon (not implemented yet, but curses makes this easy), followed by a newline.
Speed	`SPEED`	A numerical value (dice, as defined above) describing the speed of a monster, followed by a newline
Abilities	`ABIL`	List of space-delineated abilities keywords. For now, we've got `SMART`, for smart monsters, `TELE`, for telepathic monsters, `TUNNEL` for monsters that can tunnel through rock, `ERRATIC` for erratic monsters, and let's add and `PASS`, for non-corporeal monsters, like ghosts, that can pass through matter without damaging it, followed by a newline.
Hitpoints	`HP`	A numerical value describing the amount of damage a monster can take, followed by a newline.
Attack Damage	`DAM`	A numerical value describing the amount of damage a monster can inflict per attack, followed by a newline.

Ncurses allows us to display in color, assuming it's supported by the terminal. We'll define the following colors: `RED`, `GREEN`, `BLUE`, `CYAN`, `YELLOW`, `MAGENTA`, `WHITE`, `BLACK`. When we start loading (as opposed to simply reading their descriptions in from a file) these new monsters, we'll also start rendering them in color.

Let's also add some keywords to start and end each entry:

```
BEGIN MONSTER
```

on a line by itself to mark the start of a new NPC, and

```
END
```

on a line by itself to mark the end.

All fields must be present. No field may appear more than once for a given monster. On any failure (parsing error, duplicate field, missing field, etc.), the monster should be discarded and parsing should continue with the next monster in the file (if it exists). In other words, any parser error discards the current monster, and the parser scans for the next occurrence of `START MONSTER` and continues processing from there.

We'll also add some metadata to start the file for versioning:

```
RLG327 MONSTER DESCRIPTION 1
```

should be the first line. If it fails to match, you may terminate the program.

We will store our monster descriptions in the same directory where we have been storing our dungeon save files, `$HOME/.rlg327`, in a file named `monster_desc.txt`.

In order for the TAs to verify that your parser works, we'll need you to print out the monster definitions after reading them. We'll print the values of every field, one per line (the description may be printed on several lines), in the order they are given in the above table, with a blank line between each monster.

Your parser should be implemented in C++. Since we're concerned about correctly parsing your monster description files and not about the mechanics of the game, I suggest that you modify main to read the monster description files, print the test output, and exit, without ever entering the game proper.

Note that these descriptions are essentially templates for monsters, not instances of actual NPCs. Later, when we use these monster descriptions, we'll use monster descriptions to generate NPCs. So when thinking about how to represent these internally, you might consider creating a monster description class that can return instances of `npc` from a generate method. An instance of an NPC will still have dice for damage abilities (each attack is a roll), but max hitpoints and speed will be fixed based on rolls when the monster is generated.

An example of a monster description file with two entries is listed on the next page, and on the page following that, the expected output of your program on this file.

RLG327 MONSTER DESCRIPTION 1

```
BEGIN MONSTER
NAME Junior Barbarian
SYMB p
COLOR BLUE
DESC
This is a junior barbarian.  He--or is it she?  You can't tell for sure--
looks like... it should still be in barbarian school.  The barbarians are
putting them in the dungeons young these days.  It's wearing dirty, tattered
cloth armor and wielding a wooden sword.  You have a hard time feeling
intimidated.
.
SPEED 7+1d4
DAM 0+1d4
HP 12+2d6
ABIL SMART
END

BEGIN MONSTER
NAME Amazon Lich Queen
DESC
She was a powerful Amazon warrior in life.  Death at the hands of the undead
hordes was followed by her resurrection through dark, necromantic arts.  Her
power in life didn't approach her undead glory.  Clad in night-black robes
that don't move in the wind, her incorporeal form commands the power of death
over life.  You may just be her next victim.  You fear for your soul as you
quake before her malevolent majesty.
.
SYMB p
COLOR BLACK
ABIL SMART PASS
DAM 30+5d9
HP 2999+1d1001
SPEED 10+10d2
END
```

```
Junior Barbarian
This is a junior barbarian.  He--or is it she?  You can't tell for sure--
looks like... it should still be in barbarian school.  The barbarians are
putting them in the dungeons young these days.  It's wearing dirty, tattered
cloth armor and wielding a wooden sword.  You have a hard time feeling
intimidated.
p
BLUE
7+1d4
SMART
12+2d6
0+1d4

Amazon Lich Queen
She was a powerful Amazon warrior in life.  Death at the hands of the undead
hordes was followed by her resurrection through dark, necromantic arts.  Her
power in life didn't approach her undead glory.  Clad in night-black robes
that don't move in the wind, her incorporeal form commands the power of death
over life.  You may just be her next victim.  You fear for your soul as you
quake before her malevolent majesty.
p
BLACK
10+10d2
SMART PASS
2999+1d1001
30+5d9
```

Parsing Item Definitions

Optional! You don't have to do this part!

What fun is crawling through a monster-laden dungeon if all of the monsters are more powerful than you? To even out the odds, we'd like to add attributes and abilities that increase with level to the PC, but for now, let's allow the PC to pick up the leavings of adventurers past. Whenever we generate a new dungeon level, we'll also generate a few objects and scatter them around the dungeon floor. And when we kill a monster, that monster may drop items. We'll start by defining the attributes that we want in our gear and define a file format to specify them. Next week we'll start using them (and the monsters that we defined last week) in our dungeons.

Obviously a breast plate and a spiked mace should have different characteristics, so we might define a class hierarchy for all items; however, there no reason that armor cannot inflict damage on NPC or that weapons cannot grant defensive bonuses or emit light. Given this, it's premature and probably overly complex to design a polymorphic class structure for dungeon objects (though you are welcome to if either you disagree or you just want to). We can simplify things by defining only one class and deciding which fields to use based on the type of object. Before we start working out the details of objects, let's figure out what kind of equipment our PC will use.

A character can wield a weapon in hand, a shield (or off-hand weapon, or maybe a two-handed weapon) in the other hand, a bow worn on back, helmet on head, armor over body, cloak over shoulders, gloves on hands, boots on feet, rings on fingers (let's hold off on the bells on toes for now), maybe an amulet or necklace around the neck, and a light source, which gives us between 11 and 20 pieces of equipment, depending on the number of rings. In practice, in games of this type, rings tend to be fairly powerful and characters are restricted to one or two of them, so let's go with two. Below is a table listing them and giving some keywords that we'll use in the file. We'll also add some item types for this that aren't gear.

Item type	Keyword	Equipment
Main Weapon	WEAPON	✓
Shield, off-hand weapon, or two-handed weapon	OFFHAND	✓
Bow, crossbow, sling	RANGED	✓
Body armor or clothes	ARMOR	✓
Hat, helmet, or crown	HELMET	✓
Cloak	CLOAK	✓
Gloves or gauntlets	GLOVES	✓
Shoes, boots, or greaves	BOOTS	✓
Rings	RING	✓
Amulets, necklaces, or broaches	AMULET	✓
Torch, lantern, or other light source	LIGHT	✓
Scrolls, parchment, paper	SCROLL	✗
Books	BOOK	✗
Flasks, bottles, cups, maybe with something in them	FLASK	✗
Precious metals or gems, Dollars, Pounds, Euros, Kronor, RMB, or Rupees (we're international!)	GOLD	✗
Arrow, bolts, pebbles	AMMUNITION	✗
Food or drink	FOOD	✗
Wands, staves, magical implements	WAND	✗
Chests, safes, bags	CONTAINER	✗

In order to create a two-handed weapon, we'll use both the WEAPON and OFFHAND keywords, so a bitfield would be a good way to store this internally.

Just as we did for monsters, we'll define a set of objects attributes:

Parameter	Keyword	Description
Name	NAME	A name to describe the object, for instance, "a pair of leather moccasins" or "a plain gold ring".
Description	DESC	A textual description of the item, beginning on the next line after the keyword, terminated by a period on a line by itself, and limited to a width of 77 characters.
Type	TYPE	Item type keyword (above).
Color	COLOR	A color name keyword describing the color used to render the object in the dungeon. Color name keywords are as defined in assignment 1.05.
Hit bonus	HIT	A numerical offensive bonus that is applied to the probability of the wielder hitting an opponent.
Damage bonus	DAM	A numerical offensive bonus that is applied to the damage inflicted by a successful attack.
Dodge Bonus	DODGE	A numerical defensive bonus that reduces a PC's probability of being hit by an NPC attack.
Defense bonus	DEF	A numerical defensive bonus that reduces the damage incurred by a successful attack from an NPC.
Weight	WEIGHT	The object's numerical weight in *Units* (a "unit" is something that people use to measure things).
Speed bonus	SPEED	A numerical bonus to PC speed when equipping the object.
Special Attribute	ATTR	A numerical value specifying the value of any special characteristics—if any—an object may have, for instance the capacity of a container or the radius of a light source.
Value	VAL	A value in *Pesos de Ocho*.

In a similar fashion to our monster description files, we'll start the file with metadata (some semantics markers and a file version):

```
RLG327 OBJECT DESCRIPTION 1
```

And each entry will start with the line

```
BEGIN OBJECT
```

and end with the line

```
END
```

We will store our object descriptions in the same directory where we have been storing our dungeon save files and monster descriptions, `$HOME/.rlg327`, in a file named `object_desc.txt`.

In order for the TAs to verify that your parser works, we'll need you to print out the object definitions after reading them. We'll print the values of every field, one per line (the description may be printed on several lines), in the order they are given in the above table, with a blank line between each object.

Your parser should be implemented in C++. Since we're concerned about correctly parsing your object description files and not about the mechanics of the game, I suggest that you modify main to read the object description files, print the test output, and exit, without ever entering the game proper (just like we did last week).

Note that these descriptions are essentially templates for objects, not instances of actual objects. Next week, we'll use these object descriptions and last week's monster descriptions, generate objects and NPCs in the dungeon. So when thinking about how to represent these internally, keep that in mind. And just like last week, we are not actually generating instances of these things yet, just parsing them and printing them out for verification.

Beginning on the following page, we present an example object description file with several objects. We have not produced output here, because that follows directly from last weeks work, so we're confident that you can figure it out.

```
RLG327 OBJECT DESCRIPTION 1

BEGIN OBJECT
NAME a NERF(R) dagger
TYPE WEAPON
COLOR MAGENTA
WEIGHT 1+0d1
HIT 0+0d1
DAM 0+0d1
ATTR 0+0d1
VAL 9+1d6
DODGE 10+0d1
DEF 0+0d1
SPEED 0+1d0
DESC
This is a totally wicked looking dagger.  It's got awesome barbs on the back
of the blade, a compass on the hilt, and myriad other embelishments that
serve no functional purpose.  You could totally be a deadly assassin with a
sweet blade like this.  Since it's so light, it won't encumber you.  It's
made out of polyurethane foam.
.
END

BEGIN OBJECT
NAME a vorpal blade
DESC
One, two! One, two! And through and through!
It goes snicker-snack.
.
TYPE WEAPON
COLOR RED
WEIGHT 15+0d1
HIT 12+3d4
ATTR 0+0d1
VAL 198+2d51
DAM 12+3d4
DODGE 0+0d1
DEF 0+0d1
SPEED 0+1d0
END

BEGIN OBJECT
NAME a prom dress
HIT 0+2d3
DAM 0+2d3
DODGE 0+0d1
SPEED -5+0d1
ATTR 0+0d1
VAL 449+0d1
DEF 0+0d1
DESC
This dress is totes fab! It's off-the-shoulder with an A-line waste, sequins,
and lavender chiffon.  It fits tight to mid-calf and you totally can't take
```

```
full steps in it, but who cares?  You'll be the hit of the party as long as
Shirley doesn't show up wearing the same thing again.
.
TYPE ARMOR
COLOR MAGENTA
WEIGHT 0+2d2
END

BEGIN OBJECT
NAME a chainmail coif
WEIGHT 8+0d1
HIT 0+0d1
DAM 0+0d1
ATTR 0+0d1
DODGE 0+0d1
VAL 48+2d11
DEF 10+2d6
SPEED 0+1d0
DESC
A heavy chainmail head covering.
.
TYPE HELMET
COLOR BLACK
END

BEGIN OBJECT
NAME the Aegis
DESC
A very high-quality shield.  Nobody knows what it looks like, except for
maybe--now--you, but you're not telling.  The back side is inscribed with the
words: "If found, please return to Zeus, Mount Olympus".
.
TYPE OFFHAND
WEIGHT 3+0d1
HIT 0+0d1
DAM 0+0d1
VAL 17000+0d1
DODGE 20+2d8
ATTR 0+0d1
DEF 20+3d8
SPEED 10+1d0
COLOR YELLOW
END

BEGIN OBJECT
NAME a cloak of invisibility
TYPE CLOAK
VAL 10000+5d1000
DESC
One of the Deathly Hallows, this cloak grants the wearer invisibility.  Last
know to be the property of the Boy Who Lived.
.
WEIGHT 1+0d1
```

```
DODGE 24+5d4
DEF 0+0d1
HIT 0+0d1
ATTR 0+0d1
DAM 0+0d1
SPEED 0+1d0
COLOR CYAN
END

BEGIN OBJECT
NAME the One Ring
TYPE RING
WEIGHT 1+0d1
COLOR YELLOW
DODGE 50+6d8
VAL 1000000+2d1000000
DEF 50+6d8
ATTR 0+0d1
HIT 10+2d5
DAM 10+2d5
SPEED 12+1d0
DESC
The One Ring to rule them all... and in the darkness bind them.  Created by
Sauron as a tool in his quest to dominate Middle Earth.  It draws the dark
sorcerer's eye upon its wearer.
.
END

BEGIN OBJECT
NAME a torch
TYPE LIGHT
WEIGHT 2+1d2
COLOR BLACK
DODGE 0+0d1
VAL 0+1d3
DAM 0+0d1
DEF 0+0d1
HIT 0+0d1
SPEED 0+0d1
DESC
A short wooden stick topped with an oil-soaked cloth, perfect for lighting
your way through the dungeon.
.
ATTR 0+1d3
END

BEGIN OBJECT
NAME a Wicked Lasers(R) Torch
TYPE LIGHT
WEIGHT 1+1d2
COLOR BLACK
DODGE 0+0d1
VAL 199+0d1
```

```
DAM 0+0d1
DEF 0+0d1
HIT 0+0d1
SPEED 0+0d1
DESC
From the makers of the world's most refined lasers, comes the ultimate in
handheld flashlights. The Flashtorch is a compact, portable searchlight that
is capable of producing an incredible 4100 lumens of intense white light. Use
this power to guide your way home, light a fire, or even fry an egg!
.
ATTR 200+0d1
END
```