CSCI321
Computer Science Project

D20 character generation
and maintenance

Final Technical Manual

Spring Session
October 2004

'Team d20'
Caleb Avery
Mark Boxall
Mark Hellmund
Mavis Shaw
Michael Tonini

Peter Castle

Table of contents

Introduction

Welcome to the *d20 character generation and maintenance program.* This powerful tool will help you to create character statistics for all d20 game sets, as well as allowing you to make running changes throughout the life of the character. With facilities that allow you to create a new d20 system character from scratch, edit an existing character or create a random character of your choice, the *d20 character generation and maintenance program* makes character creation and maintenance easy!

Role-playing games, also known as RPG's, are paper-based adventure games that take the players on journeys and campaigns filled with monsters, treasure and knowledge, and players must create their own characters to participate in these. Based on the d20 system, players go through a series of steps and rules to create all of the characteristics required to make a character with all of the necessary attributes needed. All of this information will be stored on an appropriate medium, such as a character sheet, for future reference. An example of a popular RPG is that of Dungeons and Dragons.

Throughout game play with the character, certain aspects of the characters change due to occurring factors such as battles and discoveries. Players must continually update their character sheets to keep up with these changes, which can prove to be difficult at times, especially if a character is 'levelling up'. Many attributes and statistics affect each other, so each change that is made needs to be checked to ensure it will not impact on another. Without the appropriate tools this process can be time-consuming, but unfortunately cannot be avoided and must be completed after each campaign the character participates in. Is there an easier way to do this?

Yes there is! The *d20 character generation and maintenance program* allows you to all of this quickly and easily, and requires no pen-and-paper work to do it. For those wanting to create a new character, the program conveniently splits character generation into 10 sections for easier working. The program ensures all entered values comply with the rule set of the chosen d20 game, informing you instantly if there is a problem. *D20 character generation and maintenance program* also allows you to edit saved characters for when changes to characters statistics need to me made. Also with the built-in functionality of random character generation for a quick alternative to character creation, all of your character needs are catered for in this program.

The *d20 character generation and maintenance program* can be used for all of the popular d20 system role-playing games, including Dungeons and Dragons and d20 modern.

Project description

Title:
D20 character generation and maintenance

Description:
Role-playing games such as Dungeons and Dragons have a very complex rule set for character generation, progression and storage. The aim of this project is to create an application that embodies the current rules, plus the facility to adapt to future rule changes, that will allow the easy creation and tracking of D&D character statistics.

Extensibility to other d20-based games would be an advantage, as would other Game Administrator tools.

Tools and resources:
Operating system: any
Language: C++, no special hardware.

CSCI321 is a third year computer science subject at the University of Wollongong where a group of students work together to design and develop a computer program from initial conception to final implementation and presentation. It is an annual subject, running over two full sessions, and is normally undertaken during 3rd year studies for computer science and information technology students.

Normally, academic staff at the university that may require a system to be developed create problem descriptions for students to work with, and supervise their progress throughout the project. However, the members of *'team d20'* created and drafted a project description proposal in relation to character creation within the d20 role-playing game system. Once a suitable supervisor agreed to work with the group, the proposal was accepted as an official project.

The team is comprised of five members and one supervisor. Caleb Avery, Mark Boxall and Mark Hellmund are current third year computer science students, while Mavis Shaw and Michael Tonini are third year information and communications technology students. Peter Castle is a lecturer within the school of information and communications technology at UoW, and has offered to supervise the project. With his extensite knowledge of many RPG games his input has been invaluable.

With everyone within the group having prior knowledge of each member, the group dynamics have been strong from the start, enabling the project group to move quickly to first learn from scratch the rules of the d20 game Dungeons and Dragons, and then design and develop an effective and efficient software system that will aid players in the creation and maintenance of new and existing game characters.

Traditional character generation

Dungeons and Dragons is a d20 (20-sided die) based roll playing game (RPG). The following is a step-by-step guide to producing a 1st level character manually for playing D&D.

Prerequisites:

- o Players Handbook v.3.5 (PHB),
- o A photocopy of the character sheet (Found in the back of you PHB),
- o A pencil
- o A scrap of paper
- o Four 6-sided dice

Step 1: Check with your Dungeon Master

Check with your Dungeon Master (DM) about any house rules, and how to apply them to the standard rule set. Also find out what other members of your party have created, so that your character fits the dynamic of the party.

Step 2: Roll Ability Scores

Roll you characters six ability scores by rolling four six-sided dice. Ignore the lowest scoring die and total the three highest scoring dice. Record the six totals on your scrap of paper.

Step 3: Choose Class and Race

Choose both your character's class and race at the same time, as some classes better suit some races. (Ref. PHB 7-60.)

Step 4: Assign and adjust Ability scores

Once you have chosen your character's class and race, use the ability scores you rolled in step one and assign each to one of the six abilities: Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. Adjust your characters scores up or down, according to your race. (Ref. PHB 12.)

Step 5: Record Racial and Class Features

Your character's race and class provide certain features, such as feats and skill bonuses. Most of these are automatic, but some involve making choices and thinking ahead about upcoming character creation steps.

Step 6: Select Skills

Your character's Intelligence modifier determines how many skill points you have to 'buy' skills with. (Ref. PHB your Class). Your can buy your character's level + 3 skill ranks. So at 1st level you can buy 4 skill points in a non-cross-class skill, and 2nd level you can only have 5 ranks total in that same skill.

Step 7: Select Feat

Each 1st-level character starts with one feat. Page 90 of the PHB lists all feats, their prerequisites, and a brief description.

Step 8: Select Equipment

To randomly determine your starting gold, see page 11 of PHB. Once you have calculated this, you can then buy your own gear piece by piece, using the information from Equipment chapter of the PHB.

Step 9: Record Combat Numbers

Determine these statistics and record them on your character sheet.

Hit Points: Your hit points (HP) determine how hard your character is to kill. At 1st level, wizards and sorcerers get 4 HP; rouges and bards get 6 HP; clerics, druids, monks, and rangers get 8 HP; fighters and paladins get 10 HP; and barbarians get 12 HP. To this number, add your character's Constitution modifier.

Armor class: Your Armor Class (AC) determines how hard your character is to hit. Add the following numbers together to get your AC: 10 + your armor bonus + your shield bonus + your size modifier + your Dexterity modifier.

Initiative: Your character's initiative modifier equals your Dexterity modifier.

Attack Bonuses: Your class determines your base attack bonus. To determine your melee attack bonus for when you get into close combat fights, add your Strength modifier to your base attack bonus. To determine your ranged attack bonus for when you attack from a distance, add your Dexterity modifier to your base attack bonus.

Saving Throws: Your class determines your base saving throw bonuses. To these numbers add your Constitution modifier to get your Fortitude save, your Dexterity modifier to get your Reflex save, and your Wisdom modifier to get your Will save.

Step 10: Details

Now choose a name for your character, determine the character's gender, choose an alignment, decide the character's age and appearance and so on. (Ref Chapter 6: Description of PHB).

Elements of D&D

Abilities

There are six main abilities - Strength, Dexterity, Constitution, Intelligence, Wisdom and Charisma. Each of the abilities is a numerical representation for how strong you are, how smart you are etc. Depending on how high or low the values are for each, you will receive ability modifiers. These give you bonuses when performing certain actions. For example a stronger fighter will do more damage, as there is more power in each hit as compared to a weak individual, who would do less damage than the average person. The average for every attribute is 10. For every 2 ability points you are above or below, you take a bonus or penalty accordingly. Abilities are determined when your character is first created, then you select your class and race, then assign the highest to the appropriate ability for that class/race.

Races

D&D is a game of the imagination, as such it has multiple races in its' worlds. There are 7 races; a character can be human, dwarf, elf, gnome, half-elf, half-orc and halfling. These all affect the ability scores, for example a Dwarf has increased constitution and decreased charisma. This has an effect on what class each race is suited for, for example, a dwarf is not very charismatic, which means he is unable to negotiate as well, making people dislike him more than other races. However, he can take more hits due to his stronger constitution. This is why a dwarf is suited to be a fighter.

Classes

A character class is a chosen job/profession; with eleven choices available covering many character traits and characteristics. You can be a barbarian, bard, cleric, druid, fighter, monk, paladin, ranger, rogue, sorcerer or wizard. Different classes allow players to gain more ranks in class skills (a rogue, a born thief, can have more skill points into sleight of hand at each level than a ranger), and also what feats and skills they gain as they progress through levels. It also determines their base attack bonus, base save bonus and the maximum number of additional hit point they can gain per level.

Skills

Skills represent actions, training and disciplines in which the character is proficient. His ability to perform these at varying complexities is effected by how many skill points have been allocated to each skill – these are assigned when the character is created and are governed by their class and intelligence. Characters of certain classes and races will be more capable at certain skills - in fact some skills are class skills that give them bonuses to progress faster in them taking less skill points. For example, a wise character is better at healing, as they are more knowledgeable. A cleric can reach a higher level of healing at a lower level, as it's a class skill. Every skill has an ability where a modifier will increase or decrease its amount when a skill check is made.

Feats

Feats are very much like skills - they give a character a new ability or merely improve a skill/modifier. Unlike a skill however you do not gain levels in feats – you either have it or you don't. Feats can also have pre-requisites. Characters get one choice of a feat when they are created, then one every 3rd level thereafter.

Character description

Every character has a different physical appearance, different age, pray to a different god and have different reasons for becoming the character he or she is. Characters also have an alignment, which will determine how they act. The alignments are Lawful Good, Lawful Neutral, Lawful Evil, Neutral Good, True Neutral, Neutral Evil, Chaotic Good, Chaotic Neutral and Chaotic Evil. For example, a Chaotic Evil character will do anything that benefits them regardless of the after-effects. Whereas a Lawful Good person will uphold the law and help people at their own expense. Depending on their alignment, characters worship a god that follows that alignment. When a character is created all of the characters description details are selected, however only alignment has a direct impact on the game. A background story and looks are only for the players benefit for role-playing.

Equipment

As a character progresses they make money in the form of coins. These are in the form of copper, silver, gold or platinum pieces, which they can barter for items. Characters have armor, weapons, magical items and many miscellaneous items. Each character starts off with a different set of items depending on their class. They will also receive different amounts of starting gold, depending on their chosen class. Any modifiers will only apply to items that the character currently has equipped. Smaller creatures can only use smaller items. Characters also have weight limits depending on their strength and size – if a characters carries too much they become encumbered and move slower.

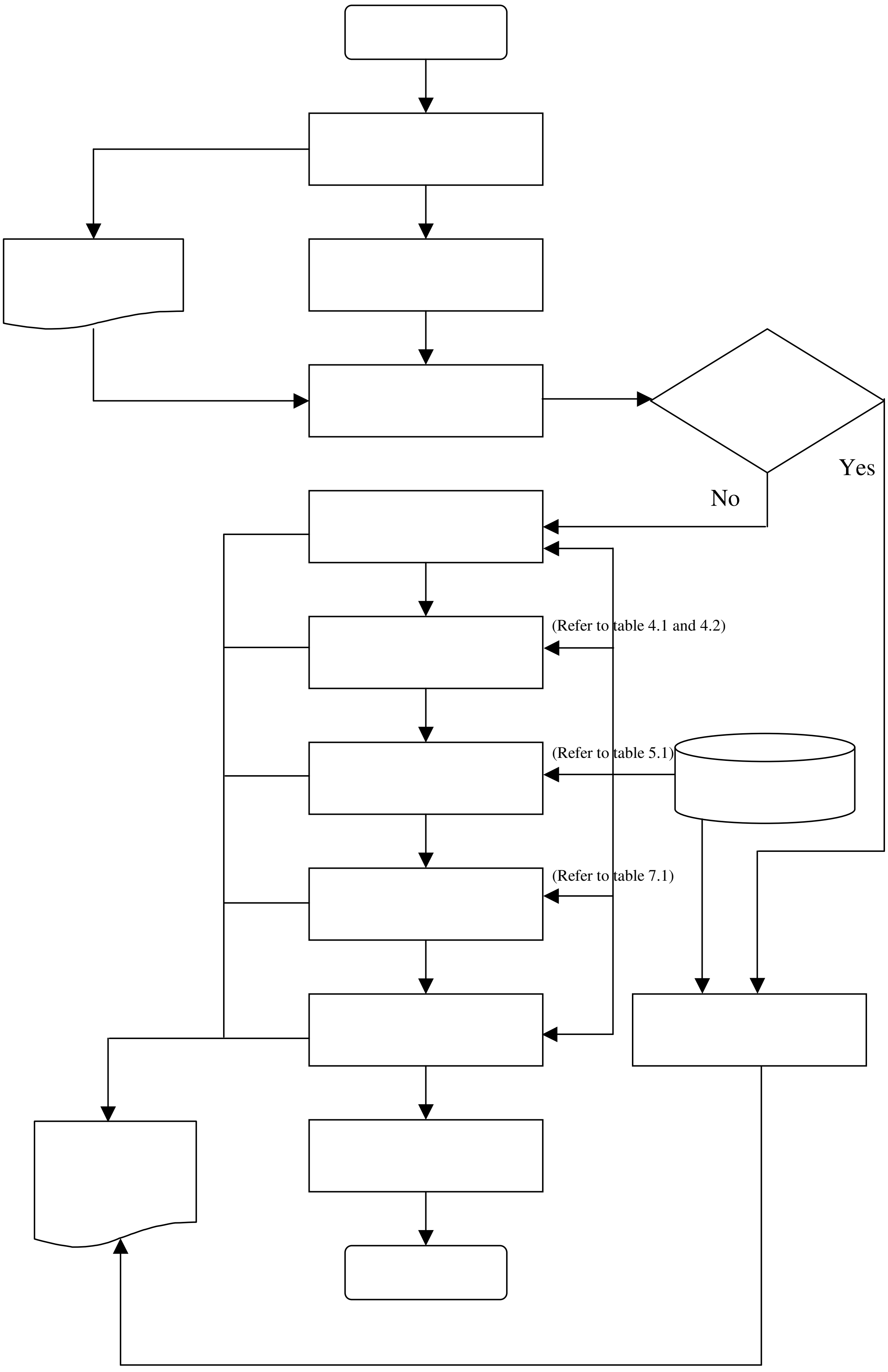
Combat

Combat in D&D is turn-based – everybody has a chance to choose their action. The order of play is decided by who has the highest initiative, this is determined by who rolls the highest number in addition to their dexterity bonus. Characters then go in the order of who attained the highest number. People then can roll for all of these actions. If their Attack Roll is higher than the monsters Armor class roll they are attacking, they score a hit and roll again for damage done. A person can use any skills they have in combat as well as perform attacks with their equipped weapon, as long as the action only uses one round. One round is determined as 6 seconds in the game world. Spells can also be cast within a round.

Adventuring

Every adventure has to start somewhere, which is usually not where it began. Journeying and exploring is a large part of D&D playing. This involves either walking, hiring a transport, riding or any of the other possibilities the player or Dungeon Master can think of. Players can go anywhere – dungeons, towns, mountains or caves – the possibilities are only limited by the Dungeon Masters imagination. Upon completing missions, treasure or payment is usually found which is divided up evenly between party members. Characters can also gain things like reputations, followers, land and titles or honours.

Character creation process



This flowchart provides a visual representation of the character creation process. There are two main sources of input/output (D&D Rulebook and Character sheet). There is also a temporary input/output to record ability rolls until they are assigned.

Processes:

- *Roll Ability Scores* – Ability scores are rolled and temporarily recorded.
- *Choose Class and Race* – Choose class and race and record them to Character sheet.
- *Assign and Adjust Ability Scores* – Based on class an race assign the ability scores to the most appropriate ability.
- *Record Racial and Class Information* – Make choices on racial and class information and record it.
- *Select Skills* – Add ranks to skills that are chosen.
- *Select Feats* – Select a feat.
- *Calculate Wealth and Select Equipment* – Roll according to class your gold amount, then select starting items.
- *Calculate Combat Numbers* -
- *Record Details*

For an in-depth description of each of the processes refer to the traditional character generation section and review the process with the corresponding title.

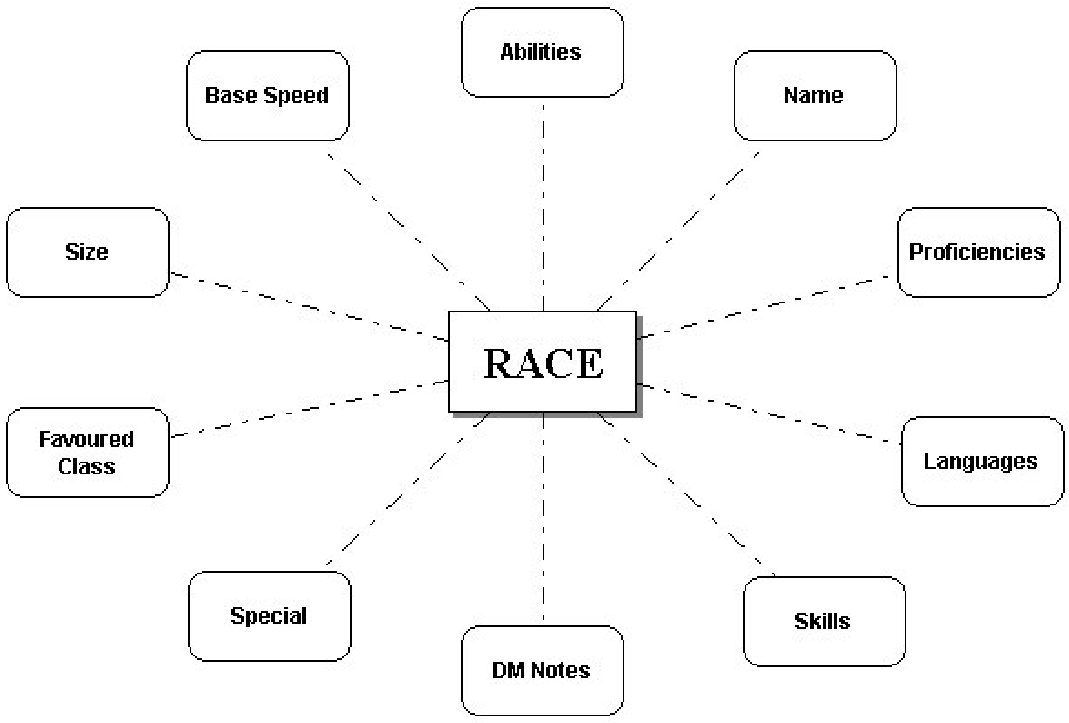
The starting package decision allows many details to be used from a predefined set instead of having to move through several of the later processes.

The processes modelled here are broken down in later sections.

Initial visual breakdown with mind maps

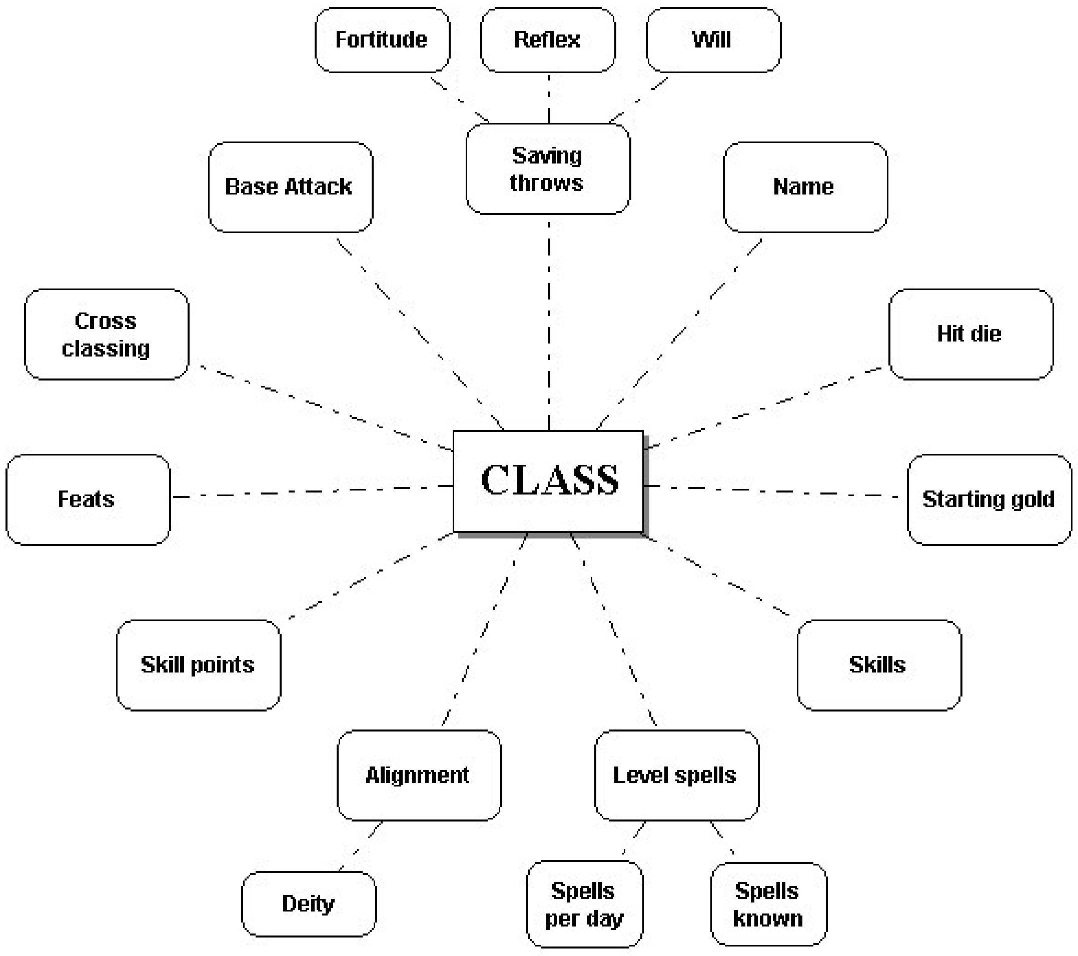
To gain a greater understanding of the elements within the Dungeons and Dragons game, mind maps were created for all of the important classes to assist in designing class structures. Through deciphering the rules the following mind maps were created, forming the basis for the attributes that would be stored within each program class. A brief description of each displayed attributes is also included.

Race



Name:	Name of the race;
Abilities:	The 6 core game elements – Strength, Dexterity, Constitution, Intelligence, Wisdom and Charisma;
Base Speed:	Moving speed of race at normal weight;
Size:	Size of the creature – tiny, small, medium, large, giant, etc;
Favoured Class:	Character class that will most suit the chosen race;
Special:	Anything special about the race that could not be categorized;
DM Notes:	Any racial information the DM will need to know:
Skills:	Skills that the race automatically gives the player;
Languages:	Languages the race knows, as well as languages they can learn;
Proficiencies:	Special circumstances where a race will get proficiencies in particular items, i.e. weapon proficiency levels may be modified for certain items.

Class

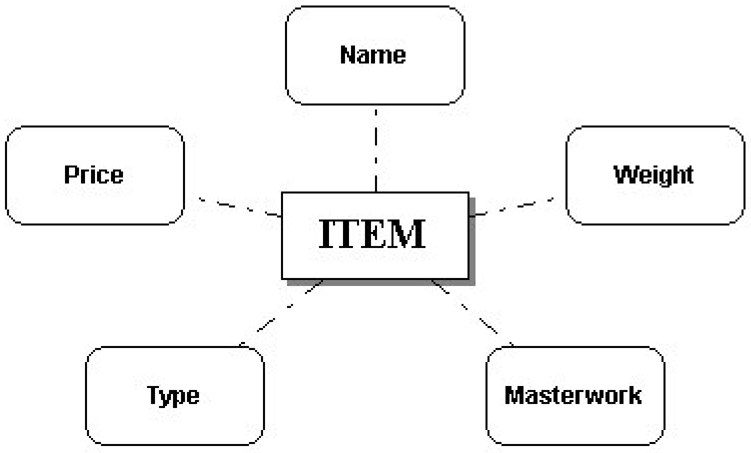


Name:	Name of class,
Hit Die:	Die used to determine attack damage,
Starting Gold:	Amount of gold allocated to character at creation,
Skills:	Skills that the class automatically gives the character,
Base Attack:	Minimum amount of damage caused in an attack,
Cross-classing:	Whether the class is affiliated with another class,
Feats:	Feats that the class automatically gives the character,
Skill Points:	Number of points class can allocate to their skills,
Saving throws:	Classes ability to withstand certain attacks, either good or bad, - Fortitude: physical stamina, - Reflex: agility or quick responses, - Will: mental thoughness,
Alignment:	Combination of attributes that determine behaviour – good, evil, lawful, chaotic, neutral, - Deity: god of worship associated with alignment,
Level spells:	 - Spells per day - Spells known

Item

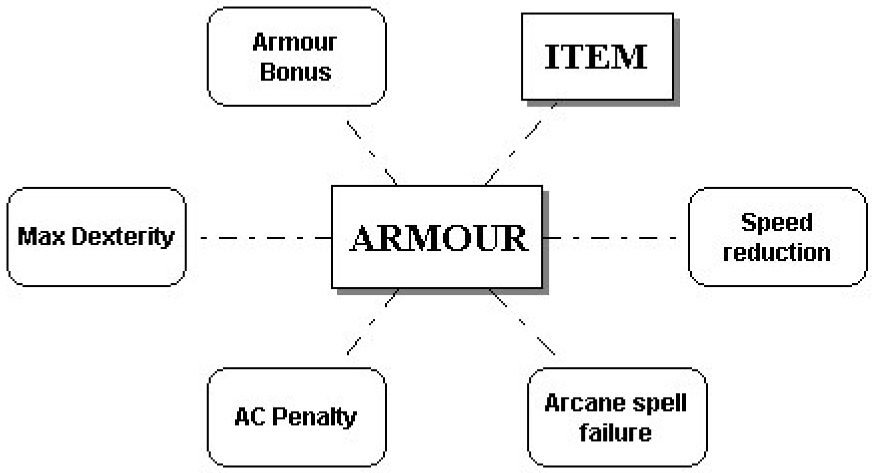
The item section proved to be an interesting one, due to the many different types of items, and how differently some of them interact with other items or characters. If all of the attributes required to cover all relevant item information were to be stored within one structure, it would be extremely inefficient due to the variation in different items.

To overcome this, a heirachy system was created where items requiring specific information to be stored about them will have a sub-class. Branching from the main items mindmap are the categories of armor, weapons and magic items. Within the items class, a flag will be set that will allocate a 0, 1, 2 or 3 depending on the type of item.



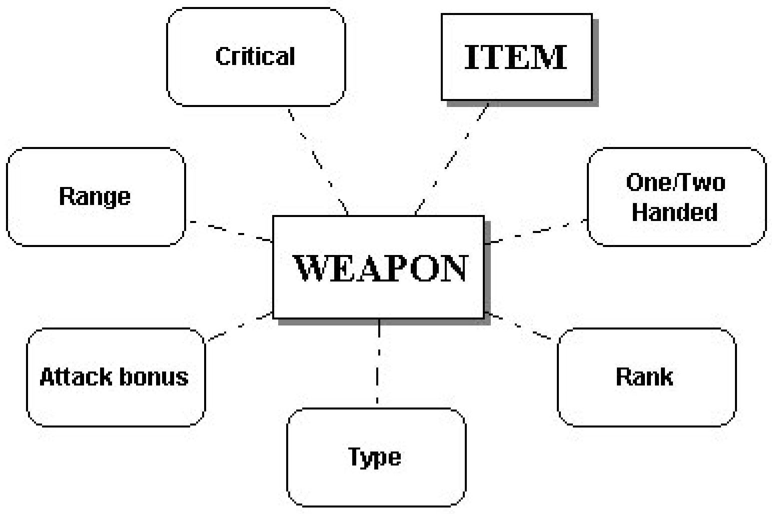
Name:	Name of item,
Weight:	Total weight of item in pounds (lb.),
Price:	Cost of item to purchase
Type:	Type of item, 0 for normal, 1 for armor, 2 for weapon, 3 for magical item
Masterwork:	Flag indicating whether item is aditionally masterworked.

Armor (inherits from Item)



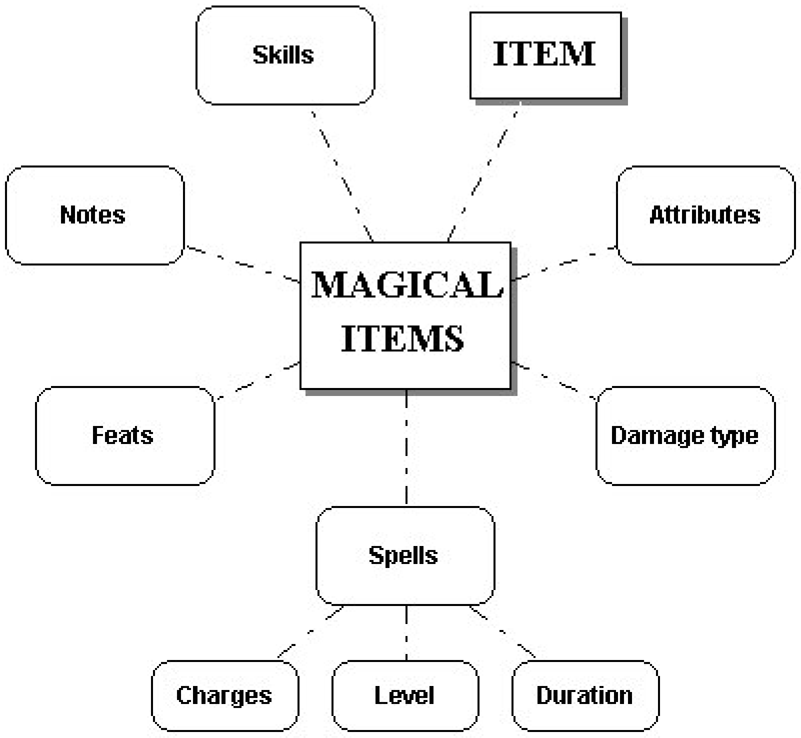
Armor bonus:	Any additional bonuses gained through armor,
Max Dexterity:	Maximum amount of Dex allowed by wearer,
AC Penalty:	AC reduction to characters attack due to weight,
Arcane spell failure:	Percentage chance of spell failing due to armor,
Speed reduction:	Maximum speed character can move wearing armor.

Weapon (inherits from Item)



Critical:	Critical roll modifier value,
Range:	Distance weapon is functional to,
Attack bonus:	Any bonuses the weapon will add to characters attack,
Type:	Main weapon category i.e. piercing,
One/two handed:	Whether one or two hands are required to use the weapon,
Rank:	Indicates skill level required to use – 0 represents simple, 1 represents martial and 2 represents exotic.

Magical items (inherits from Item)



Skills:	Skills affiliated with the magical item,
Attributes:	Attributes affiliated with the magical item,
Damage type:	Type of damage weapon will inflict,
Feats:	Feats affiliated with the magical item,
Notes:	Any special notes about the items functionality,
Spells:	Spells affiliated with the magic item,

- *Charges:* Number of spell instances remaining,
- *Level:* Caster level required to use magical item spell,
- *Duration:* Length of time spell lasts.

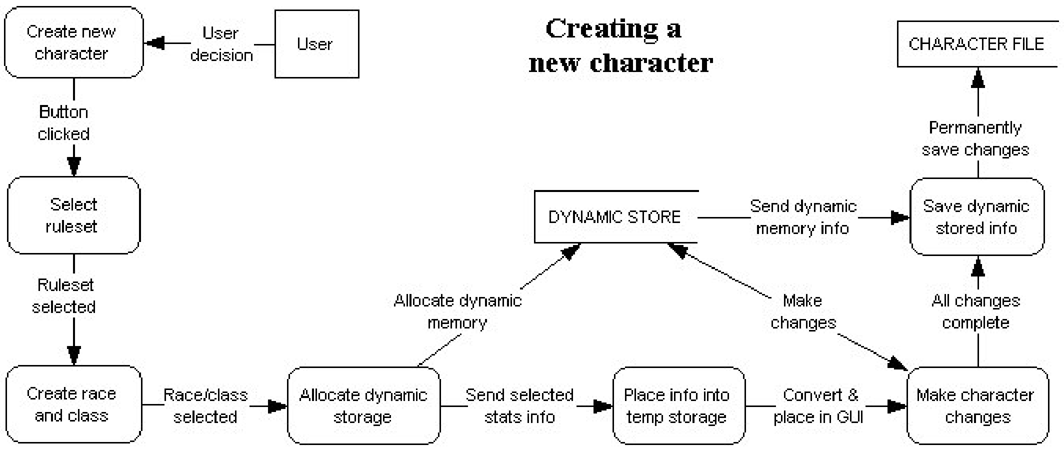
Program data process

Understanding the processes that will be undertaken in the final system will aid in the overall design process. After deciding what functions the program would perform, processes for each section were defined and converted into data flow diagrams for easier viewing and layout.

There will be three main processes within the program, which include:

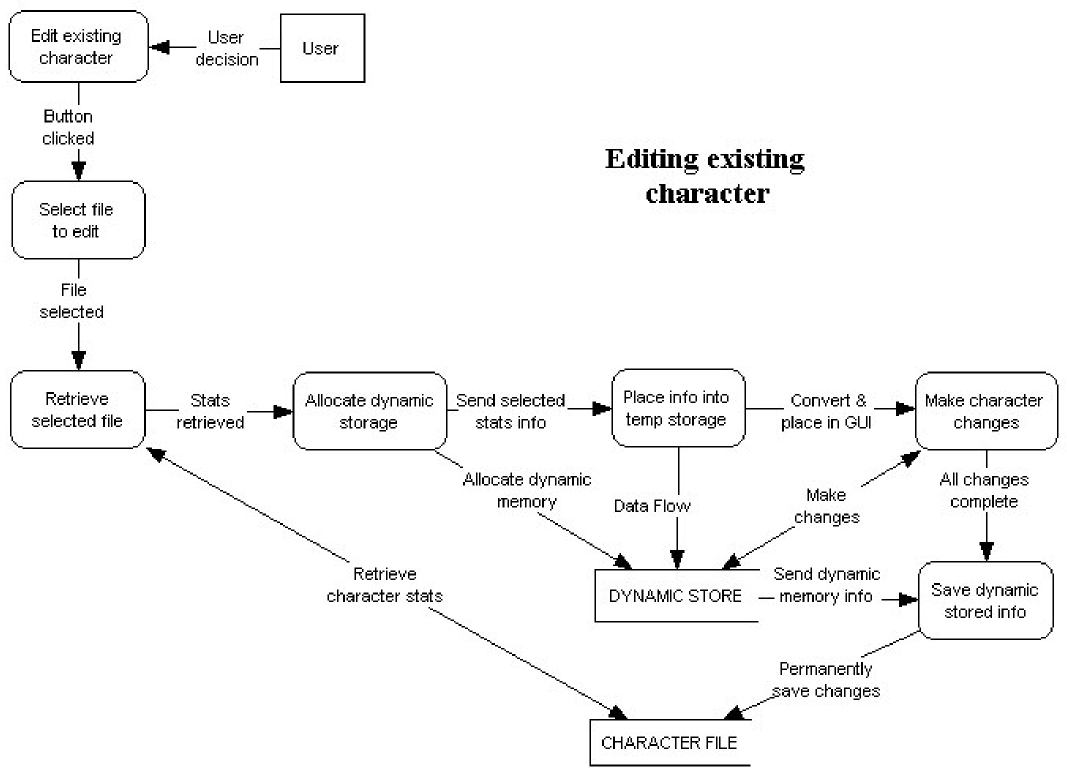
- Creating a new character
- Editing an existing character
- Random character generation

Creating a new character



- *Create new character* – User selects the program to create a new character,
- *Select ruleset* – Once selected, user chooses the ruleset they wish to use,
- *Create race and class* – Once selected, user chooses their characters race and class,
- *Allocate dynamic storage* – Once base character information is selected, new dynamic storage is allocated to store the information,
- *Place info into temp storage* – Once created, the base character information sent from earlier operations is stored within the dynamic storage, and then placed within the GUI for user manipulation,
- *Make character changes* – User makes appropriate character statistic modifications to information displayed within the GUI, which is continually updated within the dynamic store,
- *Save dynamic stored info* – Once all changes are complete, the information stored in the dynamic storage is converted into a hard copy of the character file.

Editing an existing character



- *Edit existing character* – User selects the program to edit an existing character,
- *Select file to edit* – Once selected, user finds and selects file to open,
- *Retrieve selected file* – Once selected, the character file is located and opened, ready to read in the characters information,
- *Allocate dynamic storage* – Once character information is retrieved from the file, new dynamic storage is allocated to store the information,
- *Place info into temp storage* – Once created, the base character information sent from earlier operations is stored within the dynamic storage, and then placed within the GUI for user manipulation,
- *Make character changes* – User makes appropriate character statistic modifications to information displayed within the GUI, which is continually updated within the dynamic store,
- *Save dynamic stored info* – Once all changes are complete, the information stored in the dynamic storage is converted into a hard copy of the character file.

Coding and naming conventions

Due to the large amount of people working on the project, we decided it important to create a coding and naming convention to ensure that all code developed for the project will have continuity, and allow for easier testing and collaboration of different segments of code. Following is a summary of some coding conventions that will be standard in all coding.

Block Comments

File Header:
```
/*******************************************************\
* Filename                                              *
* =========                                             *
* Project: D20 Character Generation                     *
* Created By:                                           *
* Last Edited By:                                       *
* Last Edited: 00/00/00                                 *
* Description:                                          *
*                                                       *
\*******************************************************/
```

Function Header:
```
/*******************************************************\
* function (parameters)                                 *
* parameter1 description:                               *
* parameter2 description:                               *
* ...                                                   *
* return value description:                             *
* function description:                                 *
\*******************************************************/
```

Identifiers

- multiple words separated by underscore (e.g. file_counter)
- non-constants use all lowercase
- constants are all uppercase
- identifiers must describe the function or value they perform/hold

Statement Braces

- curly braces appear on the line following the function name
- curly braces are inline with the function
- code begins one tab in from the curly brace on the next line
- switch cases appear one tab in, code to be performed appear another tab in

Identation

- Each statement block appears in a new indent

Variables

- variables defined at the beginning of the scope it is in
- variables declared in a scope inside a function should be commented with other variables for the function
- avoid global variables (except in extreme circumstances)

Editing Code

- Printing debugging information to cerr, should be highly commented to stand out
- Comment out the code to be changed, include a timestamp and the name of the person making the change, e.g.

```
// 13/03/04 Caleb
// while (1) {cin >> filename;}
for (int file_count=0;file_count<FILE_MAXFILES;file_count++)
```

cin >> filename;

Code File Names

- Classes have there own ".cpp" and ".h" files
- File names are of the form `parent_subclass.(cpp,h. (I`f a file is not part of a class, the filename must describe it's place in the program e.g. "ui_menus.h")

Pre-Compile Commands

- Use of pre-compile commands in a header file to ensure it is not included more than once, e.g.

```
#ifndef HEADER_H
 .
 .
 // Code goes here
 .
 .
#define HEADER_H
#endif
```

- Use of pre-compile commands during debugging, e.g.

```
#define DEBUG
 .
 .
 .
#ifdef DEBUG
   cerr << this_value << endl;
```

#endif

<u>Classes and implementation breakdown</u>

In order to store all of the data relevant to a D&D character in our program, a number of classes have been created to assist in the storage and manipulation of the data.

To limit the amount of hard-coded variable names and values, a mostly generic structure was created to allow for the reading of variable names and values from files. This has the added advantage of being able to use the potential variable name as an array key. Using class functions to reference values by an input string has greatly simplified the functions and allowed for a more generic approach.

The following pages will give a detailed explanation of each section of the program. There are 17 classes that comprise the data storage and manipulation of the program, and they include the following:

- char_int_group
- char_int_pair
- char_class
- class_list
- feat_list
- feats
- item_list
- items_class
- modifier_node
- money
- physical
- profile
- race
- spell
- spell_list
- utilities
- xp

Char int pair class

The *char_int_pair* class provides the overlying structure to the linked list, and is in charge of adding, removing, and changing modifiers in the list. **As there is a lot of dynamic memory, this particular class is quite vulnerable to memory leaks if the proper checks are not made.**

```
const int MODIFIER_INVALID = -1234;

class char_int_pair
{
	private:
		char* pair_name;
		modifier_node* modifier;

		int max_modifier_length;
		int num_nodes;

	public:
		char_int_pair();
		char_int_pair(char*);
		char_int_pair(const char_int_pair&);
		~char_int_pair();

		void set_name(char*);

		int get_total();
		char* get_name();

		int get_explanation_size();
		void get_explanation(char*);

		void add_modifier(char*, int);
		void remove_modifier(char*);
		void reset_modifiers();
		int get_modifier(char*);
		void set_modifier(char*, int);

		int operator[] (char*);
		char_int_pair& operator= (const char_int_pair);
		bool operator== (char*);
		bool operator!= (char*);
		bool operator< (char*);
		bool operator> (char*);
		bool operator<= (char*);
		bool operator>= (char*);
};
```

Private Members:

- *pair_name* – This is a character string that is also used as the char_int_pair's identifier.
- *modifier* – modifier_node pointer is the head of the linked list, and as such also defaults to a NULL value when first declared, and whenever the modifiers are reset.
- *max_modifier_length* – This is the length of all of the elements in the modifier list. Mainly used for memory allocation.
- *num_nodes* – This is an integer that contains the number of nodes in the char_int_pair instance.

Public Members:

- *char_int_pair()* – This default constructor sets to pointers to NULL and the intesgers to 0.
- *char_int_pair(char*)* – This initialisation constructor takes a char* that is the name of the new char_int_pair to be created. All the other values are set to NULL or 0.
- *char_int_pair(const char_int_pair&)* – This is the classes copy constructor it copies the content of a char_int_pair to the current char_int_pair.
- *~char_int_pair()* – This destructor deletes all the classes dynamic memory.
- *void set_name(char*)* – This function takes char* corresponding to the new name of the char_int_pair. This is for use mainly after using the default constructor instead of the initialisation constructor.
- *int get_total()* - This function steps through the linked list of modifiers and returns the cumulative total.
- *char* get_name()* – This function returns the name of the char_int_pair for comparison or other necessary functions.
- *int get_explanation_size()* – This function calculates the amount of dynamic memory required to store the modifier explanation and returns the result as an integer.
- *void get_explanation(char*)* – The parameter given is a char* for writing the explanation to. The char* is assumed to be of sufficient size (This size can be calculated by the get_explanation_size function described above). The function creates a string listing the modifier's names followed by sufficient padding, then a colon, a space, a sign, a two-digit number for the value and a new-line char.
- *void add_modifier(char*, int)* – The first parameter of this function is a char* containing the name of the modifier to be added. The second is an integer containing the value of the modifier, whether positive or negative. The function creates an instance of a modifier_node, that is then added to the linked list of other modifier_nodes.
- *void remove_modifier(char*)* – This function takes a char* containing the name of the modifier to be removed from the list and removes it.
- *void reset_modifiers()* – This function removes all modifier_nodes from the linked list.
- *int get_modifier(char*)* – This function takes a char* containing the name of the modifier for which the value was requested it then returns the value of the modifier requested.
- *void set_modifier(char*, int)* – The function's first parameter is char* containing the name of the modifier for which the value must change and a second parameter containing an integer expressing the new value for the modifier. The function changes the value of a modifier which matches the name of the first parameter to the value passed as the second parameter.

- *int operator[] (char*)* – This function has exactly the same function as the get_modifier function. It is here to provide a short-hand get method.
- *char_int_pair& operator= (const char_int_pair)* – This operator is overloaded so it can perform the copy constructor function.
- These operators were all overloaded so as to compare pair names easily:
  bool operator== (char);*
  bool operator!= (char);*
  bool operator< (char);*
  bool operator> (char);*
  bool operator<= (char);*
  bool operator>= (char);*

Char int group class

This class is used to create a dynamic linked list of character-integer pairs

```
class char_int_group
{
      private:
            char* group_name;
            char_int_pair* dynamic_pairs;
            int num_pairs;

            profile* parent;

            void group_quick_sort(char_int_pair*, int);
            void group_quick_pivot(char_int_pair*, int, int&);

      public:
            char_int_group();
            char_int_group(char*);
            char_int_group(profile*);

            ~char_int_group();
            void read_file(char*);

            void link_parent(profile*);

            int get_num_pairs();

            int operator[] (char*);
            int operator[] (int);

            char_int_pair* operator() (char*);
            char_int_pair* operator() (int);

            void apply_modifiers();
};
```

Private Members:

- *char* group_name* – This is a string specifying the group name.
- *char_int_pair* dynamic_pairs* – This is a pointer that is assigned memory to hold the pairs.
- *int num_pairs* - This is an integer that provides the number of pairs currently in the group.
- *profile* parent* - This provides a pointer to the parent profile class.
- *void group_quick_sort(char_int_pair*, int)* – This function sorts all of the char_int_pairs contained in the group by name.
- *void group_quick_pivot(char_int_pair*, int, int&)* – This function provides the pivot for the quick sort.

Public Members:

- *char_int_group()* – This is the default constructor that sets all pointer members to NULL.
- *char_int_group(char*)* – The char* that is passed is the file name of the file that contains the definition of the char_int_group. This initialization constructor then moves populates the group with this data.
- *char_int_group(profile*)* – This constructor takes a profile pointer that is the parent class of this instance and points the parent variable to the parent class.
- *~char_int_group()* – The destructor removes all dynamic memory.
- *void read_file(char*)* - This function takes a file specified by the char* parameter, then allocates memory for and fills the dynamic pair array with the data.
- *void link_parent(profile*)* – This function takes a profile pointer that is the parent of the current instance of the class and sets the current instance to point to the parent class.
- *int get_num_pairs()* – This function returns the number of pairs currently in the dynamic group of char_int_pairs.
- *int operator[] (char*)* - This function takes a character array as it's first parameter and uses that reference to find the specified value. If the reference is only to the pair, the return value will be the total, else the value of the modifier specified.
- *int operator[] (int)* - This function returns the total value of the char_int_pair in the array at the position specified in the int parameter.
- *char_int_pair* operator() (char*)* - This function returns a pointer to one of the char_int_pairs in the array. This is done to allow reference to the functions available to char_int_pair's.
- *char_int_pair* operator() (int)* - This is the same as the function above but using an integer instead of a char* to distinguish the correct pair.
- *void apply_modifiers()* - This function opens a file for the group which contains modifiers to be manipulated. The file is assumed to be the name of the group followed by the extension ".ddm". If this is not the case then the function will not continue but the program will continue as normal.

Character class

A characters class can affect some of the basic values on a characters sheet. A player can also choose to train in multiple classes, which makes keeping track of the changes difficult. This class was developed with the intention of having one instance per different character-class the player should choose.

```
class char_class
{
   private:
      char* name;
      char* primary_ability;
      char* special_feats;
      char* class_skills;
      int level;
      int hit_die;
      int base_attack;
      int skill_points;
      int spells_day;
      int spells_known;
      int skills;
      int feats;
      bool save_fortitude;
      bool save_reflex;
      bool save_will;

   public:
      char_class();
      ~char_class();
      void set_name(char* c_name);
      .
      .
      void set_save_will(bool c_save_will);

      char* get_name();
      .
      .
      bool get_save_will();
      bool write_file(char*);
      bool read_file(char*,char*);
};
```

Private members:

- *name* – This member is a string variable that represents the name of the character class. Once again it must remain standard throughout programming and it must be unique to the class as it is the identifying member of the class.
- *primary_ability* – This member stores a character string, which contains the current class' primary ability. A number of bonuses depend on how high the characters primary ability value is. For spell-casters this can mean extra spells per day or spells known.
- *special_feats* – This member contains all of the bonus feats that are awarded by training in the current character-class. These feats are added irrespective of how many feats the character already has.

- *class_skills* - contains all of the names of the skills that are considered relative to the current character-class. Skills not on this list cost twice as many skill points to go up a rank in.
- *level* - The current level of the character in this particular character-class is stored as an integer.
- *hit_die* – This determines the maximum number of hit points a character can go up (each time they level up) in this character-class.
- *base_attack* – This member stores the base attack bonus that a player receives due to their current character-class.
- *skill_points* – Is an integer of the total number of skill points a character has to spend.
- *spells_day* – This member represents the number of spells a character can perform per day. This member is only used if the character class allows spell casting (Bard, Cleric, Druid, Ranger, Sorcerer, and Wizard.)
- *spells_known* – This member represents total number of spells the character can learn. Again, this only applies to spell-casting classes (Bard, Cleric, Druid, Ranger, Sorcerer, and Wizard). These numbers are affected by which class a character is and by which level in that character-class a character is as well.
- *skills* – This member is an integer for the number of skills a character has.
- *feats* – This member is an integer for the number of feats a character has.
- *save_fortitude*, *save_reflex*, *save_will* These three bool types determine whether the character-class has “poor” or “good” saving throws. Once that is determined the appropriate increase in fortitude, reflex and will saves can be calculated.

<u>Public members:</u>

The large amount of functions in this class are mainly for setting and getting values so they will be described in groups.

- *char_class()* – The default constructor initialises all pointer variables to NULL and all integers to 0.
- *~char_class()* – The destructor deletes all dynamic memory.

The following is a list of the set functions of the *char_class* class. They are named such that the second part of the function name (ie. After set) is the name of the member of the class that will be modified:

```
void set_name(char*);                void set_spells_day(int);
void set_primary_ability(char*);     void set_spells_known(int);
void set_special_feats(char*);       void set_feats(int);
void set_class_skills(char*);        void set_skills(int);
void set_level(int);                 void set_save_fortitude(bool);
void set_hit_die(int);               void set_save_reflex(bool);
void set_base_attack(int);           void set_save_will(bool);
void set_skill_points(int);
```

The following is a list of get functions of the *char_class*. They are named such that the second part of the function name (ie. After get) is the name of the member that will be returned:

char get_name();*
char get_primary_ability();*
char get_special_feats();*
char get_class_skills();*
int get_level();
int get_hit_die();
int get_base_attack();
int get_skill_points();
int get_spells_day();
int get_spells_known();
int get_feats();
int get_skills();
bool get_save_fortitude();
bool get_save_reflex();
bool get_save_will();

- *bool read_file(char*,char*)* – This function takes a char* parameter which is the name of the file describing the char_class types. It also takes the class name it needs to find as the second char*. It then populates the char_class with the appropriate data. It returns a bool depending on successful read. It returns *true* for successful read and *false* if an error occurs (*bad read* or *file not found,char_class not found*).
- *bool write_file(char*)* - This function takes a char* parameter which is the name of the file that the function will write to. It writes an xml structure that is described in the char_class xml file specification. It will return *true* on a successful write and *false* if an error occurs. **NB. This function will overwrite a file if it already exists.**

char_class File Specification

The file that holds the character class data is constructed with XML, like so:

```
<d2o>
<class>
<name></name>
<primaryAbility></primaryAbility>
<specialFeats></specialFeats>
<classSkills></classSkills>
<level></level>
<hitDie></hitDie>
<baseAttack></baseAttack>
<skillPoints></skillPoints>
<spellsDay></spellsDay>
<spellsKnown></spellsKnown>
<skills></skills>
<feats></feats>
<saveFortitude></saveFortitude>
<saveReflex></saveReflex>
<saveWill></saveWill>
</class>
</d2o>
```

- *d2o* – This element specifies the bounds of the document.
- *class* – This element specifies all the data inside it belonging to one class.
- *name* – This element holds the name of the class. This is directly related to the *name* in the *char_class*. This name must be unique to all other names within the file as it is the unique identifier of the class entity.
- *primaryAbility* – This element holds a string describing the classes *primary_ability*.
- *specialFeats* – This holds a string containing the classes special feats. Each feat is delimited by a ':' as can be seen below in the Document logical layout.
- *classSkills* – This element holds a string containing the class skills. Each skill is delimited by a ':'.

- *level* – This element holds the current level of the player's character. This relates to the *level* member in the *char_class*.
- *hitdie* – This element contains a number equal to the maximum number of hit points a character of the class can go up per level.
- *baseAttack* – This element contains the *base_attack* of the class.
- *skillPoints* – This element contains the number of skill points that are still to be used by the player's character.
- *spellsDay* – This element contains an integer that specifies the number of spells can have per day.
- *spellsKnown* – This element contains an integer that specifies the number of spells the character knows.
- *skills* – This element contains the number of skills the character has
- *feats* – This element contains the number of feats a character has.
- *saveFortitude*, *saveReflex* , *saveWill* – These elements hold either a 1 or a 0 depending upon whether their saving throws are "good" (1) or "bad" (0).

<u>Document Logical Layout</u>

```
<d2o>
<class>
<name>Monk</name>
<primaryAbility>primary ability</primaryAbility>
<specialFeats>feat1:feat2</specialFeats>
<classSkills>skill1:skill2:skill3</classSkills>
<level>15</level>
<hitDie>16</hitDie>
<baseAttack>4</baseAttack>
<skillPoints>43</skillPoints>
<spellsDay>2</spellsDay>
<spellsKnown>9</spellsKnown>
<skills>5</skills>
<feats>4</feats>
<saveFortitude>1</saveFortitude>
<saveReflex>0</saveReflex>
<saveWill>1</saveWill>
</class>
</d2o>
```

name must be unique to each class entity. Both *specialFeats* and *classSkills* are multiple entries delimited by ':' as shown above.

Class_list

The class_list class takes into consideration the fact that a character can be of multiple classes:
```
class class_list
{
private:
      list_node<char_class*>* head;
      list_node<char_class*>* end;
      profile* parent;
      int count;
      int solid;
public:
      class_list();
      ~class_list();
      void clear_list();
      void link_parent(profile* new_parent);
      int get_num_classes();
      int get_num_solid();
      int get_num_unique();
      char* get_unique_class(int position);
      char* get_class(int position);
      char_class* get_pointer(int position);
      char_class* get_pointer(char* name);
      void add_class(char_class* new_class);
      int get_total_level();
      char_class* get_head();
      char_class* get_tail();
      int get_num_specific_class(char* name);
      void solidify();
      void remove_from_tail();
      bool is_solid();
};
```

<u>Private Members:</u>

- *list_node<char_class*>* head* – This is the head of the list that will be constructed in this class.

- *list_node<char_class*>* end* - This is the end of the list that will be constructed in this class.

- *profile* parent* – This provides storage for a pointer to the parent profile class.

- *int count* – This member is a count of the number of classes in the list.

- *int solid* - This member is used to "solidify" the list. Solidify meaning that any item contained within the present list cannot be removed.

<u>Public Members:</u>

- *class_list()* – This default constructor sets all pointer members to NULL and all integer values to 0.

- *~class_list()* – This destructor clears all dynamic memory for the class and clears the list.

- *void clear_list()* – This function is used to clear the list that this class builds.

- *void link_parent(profile* new_parent)* – This function contains a profile pointer parameter that will link this instance of the class to that pointer.

- *int get_num_classes()* – This function returns the number of classes that are currently in the list.

- *int get_num_solid()* – This function returns the value of solid.

- *int get_num_unique()* – This function returns the number of unique class names found in the list are in the list.

- *char* get_unique_class(int position)* – This function takes the position in the list as a parameter and returns the relevant unique class name from the unique names in the list.

- *char* get_class(int position)* - This function takes the position in the list as a parameter and returns a class name from that position.

- *char_class* get_pointer(int position)* – This function takes a position in the list as it's parameter and returns a pointer to a char_class instance in the list position.

- *char_class* get_pointer(char* name)* – This function takes a char* as class name and returns a pointer to an instance of a char_class within the list.

- *void add_class(char_class* new_class)* – This function takes a new char_class as it's parameter and adds it to the list.

- *int get_total_level()* – This function returns the number of classes currently in the list.

- *char_class* get_head()* – This function returns the name of the class at the head of the list.

- *char_class* get_tail()* – This function returns the name of the class at the tail of the list.

- *int get_num_specific_class(char* name)* – This function takes the name of a class as a parameter and returns the number it is positioned in within the class list.

- *void solidify()* - This function "solidifies" the list by assigning the list's count to it's solid member. In other words, it locks down the list so that none of the current items can be removed from the list.

- *void remove_from_tail()* – This function removes the last element from the tail of the list and discards it.

- *bool is_solid()* – This returns true if count and solid are equal and false otherwise.

Feats

The *feat* class is used to apply any bonuses (modifiers) that are received for having a particular feat. It is a class that stores individual feat information for the current character.

```
class feat
{
      private:
            char* name;
            modifier_node* effects;
            feat_list* parent_list;

      public:
            feat();
            feat(char*);
            feat(feat_list*);
            ~feat();
            void use_description(char*);
            void apply_modifiers();
            void clear_lists();
            char* get_name();
};
```

Private Members:

- *name* – This is a char* that holds the name of the feat.
- *effects* – This holds a modifier_node pointer that is used to store a linked list of effects the feat has.
- *parent_list* – This is a feat_list pointer that points to a list that contains the other feats that the character has. (See feat_list class for more information.)

Public Members:

- *feat()* - Default constructor sets all the private members to NULL.
- *feat(char*)* – The char* parameter of this initialization constructor takes the name and all of the values of the feat. The line that is passed in is of the format: *featname:modifier[modifier_value]*
- *feat(feat_list*)* – Another initialisation constructor. This takes a *feat_list** that is the pointer to the list that contains the instance of the feat. It will link to the feat_list containing this instance and it will set the other values to NULL.
- *~feat()* – This destructor clears any lists and deletes dynamic memory.
- *void use_description(char*)* – The char* parameter of this function takes the name and all the values of the feat. The line that is pasted is in the same format as that of the initialisation constructor (see above).
- *void apply_modifiers()* – This function takes the *effects* member. If the parent_list isn't a NULL value then the modifiers from *effects* are sent to the parent list for processing.

- *void clear_lists()* – This function removes all entries from the lists and clears dynamic memory.
- *char* get_name()* – This function returns the char* *name* of the feat.

Feat list

The feat_list class is a linked list of individual feats so-as to provide a method for manipulating the list.

```
class feat_list
{
      private:
            list_node<feat*>* head;
            profile* parent;

      public:
            feat_list();
            feat_list(profile*);
            ~feat_list();
            void link_parent(profile*);
            void clear_list();
            void apply_feats();
            void apply_modifiers(modifier_node* head);
            void add_feat(char* description);
            feat* operator() (int);
            feat* operator() (char*);
            char* operator[] (int);
};
```

Private Members:

- *head* – This is a list_node that is used to perform operations in the *feat*_list class.
- *parent* – This is a pointer to the profile class so they are able to communicate rules between each other.

Public Members:

- *feat_list()* – This default constructor sets the two pointer members to NULL.
- *feat_list(profile*)* – This contains a pointer that points to the parent class and it set head to null.
- *~feat_list()* – This destructor clears the list and deletes dynamic memory.
- *void link_parent(profile*)* – The first parameter is a profile pointer to the parent class. It sets a link between the parent class and the current instance.
- *void clear_list()* – This clears the list and deletes dynamic memory.
- *void apply_feats()* – This function steps through the feats list and applies the modifiers that each feat contains.
- *void apply_modifiers(modifier_node* head)* – This function has a parameter that is a modifier_node pointer which is the head of a list of modifiers passed from a feat in order for the modifiers to take effect. It applies the modifiers from a feat, passing them further back to the parent class if needed, in order to apply bonuses.

- *void add_feat(char* description)* – The parameter passed is a line that contains all the data of the feat to be added. This is in the format *name:modifier[modifier_value]*.
- *feat* operator() (int)* – The parameter of this operator is an integer indicating the position in the list of the requested item. It returns a pointer to the feat in the list that appears at the position given as by the parameter.
- *feat* operator() (char*)* – The parameter of this operator is a char* indicating the name of the feat that is being requested. It returns a pointer to the feat in the list that has the name specified in the parameter.
- *char* operator[] (int)* – The parameter of this operator is an integer indicating the position in the list of the requested item. It returns a pointer to the name of a feat in the list which appears at the position given as the parameter passed.

Item Lists

For dungeons and dragons it was necessary to be able to store many items. Due to the vastly different types of items and the many different type of variables needed to store them it was necessary to create multiple lists for each item type.

Every item list based class is comprised of one centre variable. A map, which uses the item name as a key (which is a string), then a pair variable that stores an instance of the item, along with an integer holding how many of that item the character has. This is so a character with 1000 arrows for instance, would mean we had to store 1000 instances of a class with arrow stats.

All the lists also required that the STL map class be a friend. Essentially each item list handles all contact with the map class. A pair with is all that is needed to pass items to be added to the map. To parse through stored items the application merely need to create an iterator and functions were added to return iterators to the beginning and end of the map. Each class also overloads operator[], which can used to search for a specific item.

Compare Function

A map by default, which attempt to sort its list of item as they're entered. By default however it orders char*'s by their pointer value, this struct was required to make it dereference and compare the value of the strings.

```
struct ltstr
{
	bool operator()(char* s1, char* s2) const
	{
		return strcmp(s1,s2) <0;
	}
};
```

Item List Class

This is a class to handle storing the most basic of items. Write all items in the map to a file and also read in values from a text file. All other classes to handle item lists started as this class then were expanded.

```
class item_list
{
   private:
      map<char*,pair<int,item*>,ltstr >* itemlist;
      friend class map;

   public:
      item_list();
      ~item_list();
      pair<int,item*> operator[] (char* thekey);

      void add_item(pair<char*,pair<int,item*> > i_pair);
      bool remove_item(char* i_name);
      void clear();

      map<char*,pair<int,item*>,ltstr >::iterator begin();
      map<char*,pair<int,item*>,ltstr >::iterator end();

      void write_to_file(char* filename);
      bool read_from_file(char* filename);

      void output();
};
```

<u>Private Member :-</u>

- *map<char*,pair<int,item*>,ltstr >* itemlist* - A pointer to the actual map that contains items. Each map entry contains a key, which is the item name, the data the map holds is a pair. The pair comprises of an integer representing how many of each item there is, then the second part of the pair is the item instance itself. The string name in the key is actaully a pointer to the item name within the item class. ltstr is a compare function needed by the map to sort the keys. Without it map would be sorting the list by pointer value, not the actual string.
  A pointer is used as a item list could get rather large, so instead of passing a copy of the whole map by reference a pointer is used to reference the map.

<u>Public Members :-</u>

- *item_list()* - A constructor that creates the map instance.
- *~item_list()* - A destructor that deallocates the memory for the item in the pair. Then clears the map and deletes it.
- *pair<int,item*> operator[] (char* thekey)*
  Overloaded index operator, it will return a copy of the data in the map that has the key.

- *void add_item(pair<char*,pair<int,item*> > i_pair)*
  This is a function to add a pair to the map. One of the problems encountered was that there were actaully 2 keys required to store items. Generally an item has a given set of values that never change, the only exception however is masterwork. The add function used to check if an item was in the list by searching on the item name, if it found it, rather than copying another instance in, it merely incremented the counter of those items. The problem was that an item that was masterwork would be stacked with items that weren't. To overcome this add_item will actaully check if an item is masterwork, then create a mangled item name, with master in front. This way masterwork items are stored in the same list but are separate entities.

- *bool remove_item(char* i_name)*
  This is a simple remove function, a name of an item is sent, if it can be found then it is deleted and a true value is returned. Else a false value is returned if the item cannot be found.

- *void clear()*
  This is a function that can remove all items in the map, it de-allocated all dynamic memory then clears the map.

- *map<char*,pair<int,item*>,ltstr >::iterator begin()*
  This is a simple function that returns an iterator to the start of the map for parsing through all map values

- *map<char*,pair<int,item*>,ltstr >::iterator end()*
  This is also a simple function that returns an iterator to the end of the map

- *void write_to_file(char* filename)*
  This function write all data currently stored in the map to a file by the name filename. It will overwrite any file currently by that name. There are no varying size variables, therefore it simple ordered write to file.

- *bool read_from_file(char* filename)*
  This function reads in data from a text file filename into the map, if file opening fails it will return false. If the file was opened and the function reads to end of file it will return true. All the values are read in, converted to numerical value is neccessary then added to instances of item class. That is then inserted into a pair, which is passed to the add_item function.

- *void output()*
  A test function, this outputs all data currently entered into the map.

File Storage Structure

All are simple datatypes, all data is static hence simple

Tally Integer
Base Material String
Currency String
Description String
Master Work Bool Value
Item Name String
Price Integer
Type String
Weight Integer

The orders is:

- *Tally* - This is a counter of how of this kind of item the character possess. In this instance it is "1".
- *Base Material* - The base material the weapon is compromised of. In this instance the base material is set to "base material".
- *Currency* - The kind of currency needed to purchase the item. It is currently set to "currency".
- *Description* - A simple string description of the item.
- *Masterwork* - A flag as to whether an item is masterwork or not.
- *Item Name* - The main key of the item, it's unique name.
- *Price* - The number, in the currency above required to buy the item.
- *Type* - A string to represent the type of item it is.
- *Weight* - A number that represent how many pounds somethign weighs.

Armor List Class

This is a class to handle storing the most basic of armors. Write all armors in the map to a file and also read in values from a text file. This is an extended version of item list, all function have the same names but read/write to file and add armor are different, as well as the default constructor and destructor.

```
class armor_list
{
    private:
       map<char*,pair<int,armor*>,ltstr >* armorlist;
       friend class map;

    public:
       armor_list();
       ~armor_list();
       pair<int,armor*> operator[] (char* thekey);

       void add_armor(pair<char*,pair<int,armor*> > i_pair);
       bool remove_armor(char* i_name);
       void clear();

       map<char*,pair<int,armor*>,ltstr >::iterator begin();
       map<char*,pair<int,armor*>,ltstr >::iterator end();

       void write_to_file(char* filename);
       bool read_from_file(char* filename);

       void output();
};
```

Private Member :

- *map<char*,pair<int,armor*>,ltstr >* armorlist* - A pointer to the actual map that contains armors. Each map entry contains a key which is the armor name, the data the map holds is a pair. The pair comprises of an integer representing how many of each item there is, then the second part of the pair is the armor instance itself. The string name in the key is actaully a pointer to the armor name within the item class. ltstr is a compare function needed by the map to sort the keys. Without it map would be sorting the list by pointer value, not the actual string. A pointer is used as a armor list could get rather large, so instead of passing a copy of the whole map by reference a pointer is used to reference the map.

Public Members :

- *armor_list()* - A constructor that creates the map instance.
- *~armor_list()* - A destructor that deallocates the memory for the armor in the pair. Then clears the map and deletes it.
- *pair<int,armor*> operator[] (char* thekey)*
  Overloaded index operator, it will return a copy of the data in the map that has the key.

- *void add_armor(pair<char*,pair<int,armor*> > i_pair)*
  This is a function to add a pair to the map. One of the problem encountered was that there was actaully 2 keys required to store armors. Generally an armor has a given set of values that never change, the only exception however is masterwork. The add function used to check if an armor was in the list by searching on the armor name, if it found it, rather than copying another instance in, it merely incremented the counter of those armors. The problem was that an armor that was masterwork would be stacked with armors that weren't. To overcome this add_armor will actaully check if an armor is masterwork, then create a mangled armor name, with master in front. This way masterwork armors are stored in the same list but are separate entities.

- *bool remove_armor(char* i_name)*
  This is a simple remove function, a name of an armor is sent, if it can be found then it is deleted and a true value is returned. Else a false value is returned if the armor cannot be found.

- *void clear()*
  This is a function that can remove all armors in the map, it de-allocated all dynamic memory then clears the map.

- *map<char*,pair<int,armor*>,ltstr >::iterator begin()*
  This is a simple function that returns an iterator to the start of the map for parsing through all map values

- *map<char*,pair<int,armor*>,ltstr >::iterator end()*
  This is also a simple function that returns an iterator to the end of the map

- *void write_to_file(char* filename)*
  This function writes all data currently stored in the map to a file by the name filename. It will overwrite any file currently by that name. There are no varying size variables, therefore it simple ordered write to file.

- *bool read_from_file(char* filename)*
  This function reads in data from a text file filename into the map, if file opening fails it will return false. If the file was opened and the function reads to end of file it will return true. All the values are read in, converted to numerical value is neccessary then added to instances of armor class. That is then inserted into a pair, which is passed to the add_armor function.

- *void output()*
  A test function, this outputs all data currently entered into the map.

File Storage Structure
All are simple datatypes, all data is static hence simple

Tally Integer
Base Material String
Currency String
Description String
Master Work Bool Value
Item Name String
Price Integer
Type String
Weight Integer
Armor Type String
Armor Size String
Armor Bonus int value
Armor Max Dexterity int value
Armor Check Penalty int value
Armor Arcane Failure int value
Armor Speed 20 int value
Armor Speed 30 int value

The orders is:

- *Tally* - This is a counter of how of this kind of item the character possess. In this instance it is "3".
- *Base Material* - The base material the weapon is compromised of. In this instance the base material is set to "Steel".
- *Currency* - The kind of currency needed to purchase the item. It is currently set to "gold coins".
- *Description* - A simple string description of the item. Currently set to "Medium Armor".
- *Masterwork* - A flag as to whether an item is masterwork or not. Currently set to "1" meaning true.
- *Item Name* - The main key of the item, it's unique name. Currently set to "Master Scale Mail".
- *Price* - The number, in the currency above required to buy the item. Currently set to "50".
- *Type* - A string to represent the type of item it is. Currently set to "Armor".
- *Weight* - A number that represent how many pounds somethign weighs. Currently set to "30".
- *Armor Type String* - String that describes the type of armor it is. Currently set to "Medium".

- *Armor Size String* - The size class that the armor belongs to.
- Currently set to "Large".

- *Armor Bonus int value* - The bonus to Armor Class the armor gives.
- Currently set to "4".

- *Armor Max Dexterity int value* - The maximum bonus that a character dexterity bonus can give to armor class with this kind of armor. Currently set to "3".

- *Armor Check Penalty int value* - A penalty that applies to skill checks when wearing this armor. Currently set to "3".

- *Armor Arcane Failure int value* - A percentage base chance of failing at casting a spell when wearing this armor Currently set to "25".

- *Armor Speed 20 int value* - The distance a character than can move at an unencumbered speed of 20ft can move at when wearing this armor. Currently set to "15".

- *Armor Speed 30 int value* - The distance a character than can move at an unencumbered speed of 20ft can move at when wearing this armor. Currently set to "15".

Weapon List Class

This is a class to handle storing the most basic of weapons. Write all weapons in the map to a file and also read in values from a text file. This is an extended version of item list, all function have the same names but read/write to file and add weapon are different, as well as the default constructor and destructor.

```
class weapon : public item
{
    public:
       weapon();
       ~weapon();
       char* get_weapon_critical();
       .
       .
       bool get_weapon_strength_bonus();

       void set_weapon_critical(char* w_weapon_critical);
       .
       .
       void set_weapon_strength_bonus(bool w_weapon_strength_bonus);

    protected:
       char* weapon_critical;
       char* weapon_damage;
       char* weapon_type;
       char* weapon_handedness;
       char* weapon_size;
       int weapon_rank;
       int weapon_range;
       bool weapon_two_handed_bonus;
       bool weapon_strength_bonus;
};
```

Private Member :-

- *map<char*,pair<int,weapon*>,ltstr >* weaponlist* - A pointer to the actual map that contains weapons. Each map entry contains a key which is the weapon name, the data the map holds is a pair. The pair comprises of an integer representing how many of each weapon there is, then the second part of the pair is the weapon instance itself. The string name in the key is actaully a pointer to the weapon name within the item class. ltstr is a compare function needed by the map to sort the keys. Without it map would be sorting the list by pointer value, not the actual string. A pointer is used as a weapon list could get rather large, so instead of passing a copy of the whole map by reference a pointer is used to reference the map.

Public Members :

- *weapon_list()* - A constructor that creates the map instance.
- *~weapon_list()* - A destructor that deallocates the memory for the weapon in the pair. Then clears the map and deletes it.

- *pair<int,weapon*> operator[] (char* thekey)*

  Overloaded index operator, it will return a copy of the data in the map that has the key.

- *void add_weapon(pair<char*,pair<int,weapon*> > i_pair)*

  This is a function to add a pair to the map. One of the problem encountered was that there was actaully 2 keys required to store weapons. Generally a weapon has a given set of values that never change, the only exception however is masterwork. The add function used to check if an weapon was in the list by searching on the weapon name, if it found it, rather than copying another instance in, it merely incremented the counter of those weapons. The problem was that an weapon that was masterwork would be stacked with weapons that weren't. To overcome this add_weapon will actaully check if an weapon is masterwork, then create a mangled weapon name, with master in front. This way masterwork weapons are stored in the same list but are separate entities.

- *bool remove_weapon(char* i_name)*

  This is a simple remove function, a name of an weapon is sent, if it can be found then it is deleted and a true value is returned. Else a false value is returned if the weapon cannot be found.

- *void clear()*

  This is a function that can remove all weapons in the map, it de-allocated all dynamic memory then clears the map.

- *map<char*,pair<int,weapon*>,ltstr >::iterator begin()*

  This is a simple function that returns an iterator to the start of the map for parsing through all map values

- *map<char*,pair<int,weapon*>,ltstr >::iterator end()*

  This is also a simple function that returns an iterator to the end of the map

- *void write_to_file(char* filename)*

  This function write all data currently stored in the map to a file by the name filename. It will overwrite any file currently by that name. There are no varying size variables, therefore it simple ordered write to file.

- *bool read_from_file(char* filename)*

  This function reads in data from a text file filename into the map, if file opening fails it will return false. If the file was opened and the function reads to end of file it will return true. All the values are read in, converted to numerical value is neccessary then added to instances of weapon class. That is then inserted into a pair, which is passed to the add_weapon function.

- *void output()*

  A test function, this outputs all data currently entered into the map.

File Storage Structure
All are simple datatypes, all data is static hence simple

Tally Integer
Base Material String
Currency String
Description String
Master Work Bool Value
Item Name String
Price Integer
Type String
Weight Integer
Weapon critical String
Weapon damage String
Weapon type String
Weapon handedness String
Weapon size String
Weapon rank Int
Weapon range Int
Weapon two_handed_bonus Bool
Weapon strength_bonus Bool

The orders is:

- *Tally* - This is a counter of how of this kind of item the character possess. In this instance it is "1".
- *Base Material* - The base material the weapon is compromised of. In this instance the base material is set to "Steel & wood".
- *Currency* - The kind of currency needed to purchase the item. It is currently set to "coins".
- *Description* - A simple string description of the item. Currently set to "light melee weapon".
- *Masterwork* - A flag as to whether an item is masterwork or not. Currently set to false(0).
- *Item Name* - The main key of the item, it's unique name. The name is currently set to "kama".
- *Price* - The number, in the currency above required to buy the item. Currently set to 2.
- *Type* - A string to represent the type of item it is. Currently set to "weapon".
- *Weight* - A number that represent how many pounds somethign weighs. Currently set to 2.

- *Weapon critical* - A string representing how many extra rolls the player gets when they score a critical with this weapon. It's currently set to "2x".
- *Weapon damage* - A string representing what kind of damage the weapon does, Currently set to "Slashing" damage.
- *Weapon type* - String representing what kind of weapon it is. The current weapon is a "light melee weapon".
- *Weapon handedness* - A string depicting whether an item is one handed, two handed etc. Currently set to "One handed".
- *Weapon size* - A String representing the size categorie of the weapon. Currently it's set to "small".
- *Weapon rank* - A integer of the rank of the weapon. Currently set to 5.
- *Weapon range* - An integer representing the maximun range of a weapon before you take range penalties.
- *Weapon two_handed_bonus* - A bool representing whether the weapon has a two handed bonus. Currently set to false.
- *Weapon strength_bonus* - A bool depicting whether an item can add strength bonus to weapon damage.

Magical Item List Class

Magical Item list is an extended version of item list. It contains all the functions, the only real changes were to the file output and input functions, along with the constructor and destructor. All magical items are also masterwork.

```
class magical_item_list
{
private:
	map<char*,pair<int,magical_item*>,ltstr >* magical_itemlist;
	friend class map;

public:
	magical_item_list();
	~magical_item_list();
	pair<int,magical_item*> operator[] (char* thekey);

	void add_magical_item(pair<char*,pair<int,magical_item*> > i_pair);
	bool remove_magical_item(char* i_name);
	void clear();

	map<char*,pair<int,magical_item*>,ltstr >::iterator begin();
	map<char*,pair<int,magical_item*>,ltstr >::iterator end();

	void write_to_file(char* filename);
	bool read_from_file(char* filename);

	void output();
};
```

Private Member :-

- *map<char*,pair<int,magical_item*>,ltstr >* magical_item_list* - A pointer to the actual map that contains magical items. Each map entry contains a key which is the item name, the data the map holds is a pair. The pair comprises of an integer representing how many of each item there is, then the second part of the pair is the item instance itself. The string name in the key is actaully a pointer to the item name within the item class. ltstr is a compare function needed by the map to sort the keys. Without it map would be sorting the list by pointer value, not the actual string. A pointer is used as a item list could get rather large, so instead of passing a copy of the whole map by reference a pointer is used to reference the map.

Public Members :-

- *magical_item_list()* - A constructor that creates the map instance.
- *~magical_item_list()* - A destructor that deallocates the memory for the item in the pair. Then clears the map and deletes it.
- *pair<int,magical_item*> operator[] (char* thekey)*
  Overloaded index operator, it will return a copy of the data in the map that has the key.

- *void add_magical_item(pair<char*,pair<int,magical_item*> > i_pair)*
  This is a function to add a pair to the map. The problem of masterworkd is not an issue with magical items as all magical items have to be masterwork.So add_magical_item just checks if it exists, if it does it increments it.
- *bool remove_magical_item(char* i_name)*
  This is a simple remove function, a name of an item is sent, if it can be found then it is deleted and a true value is returned. Else a false value is returned if the item cannot be found.
- *void clear()*
  This is a function that can remove all items in the map, it de-allocated all dynamic memory then clears the map.
- *map<char*,pair<int,magical_item*>,ltstr >::iterator begin()*
  This is a simple function that returns an iterator to the start of the map for parsing through all map values
- *map<char*,pair<int,magical_item*>,ltstr >::iterator end()*
  This is also a simple function that returns an iterator to the end of the map
- *void write_to_file(char* filename)*
  This function write all data currently stored in the map to a file by the name filename. It will overwrite any file currently by that name. There are no varying size variables, therefore it simple ordered write to file.
- *bool read_from_file(char* filename)*
  This function reads in data from a text file filename into the map, if file opening fails it will return false. If the file was opened and the function reads to end of file it will return true. All the values are read in, converted to numerical value is neccessary then added to instances of item class. That is then inserted into a pair, which is passed to the add_item function.
- *void output()*
  A test function, this outputs all data currently entered into the map.

File storage

Magical Item contains 2 classes and multiple vectors and char_int_pairs. This makes storage more complicated as any item can have an infinite number of spells or none. It can be intelligent or not and also each vector coudl be of tremendous size. To define fields a hash("#") was used a segment delimiter. If the item possesses intelligence it also has to store multiple vectors for the langauges it knows etc.

File Storage Structure

Tally Integer
Base Material String
Currency String
Description String
Master Work Bool Value
Item Name String
Price Integer
Type String
Weight Integer
Feats Vector<char*>
Notes Vector<char*>
Cursed Notes Vector<char*>
Skills Char_int_pair
abilities Char_int_pair
combat_stats Char_int_pair
Intelligence item_intelligence:-
- Alignment String
- Communication String
- Senses String
- Purpose String
- Languages Vector<char*>
- Lesser Powers Vector<pair<char*,long> >
- Greater Powers Vector<pair<char*,long> >
- Dedicated Powers Vector<pair<char*,long> >
- Intelligence Integer
- Wisdom Integer
- Charisma Integer
- Num Lesser Integer
- Num Greater Integer
- Ego Integer
- Intel Cost Long
- Read Langauge Boolean
- Read Magic Boolean

Spell List Vector<pair<char*,int> >

The orders is:

- *Tally* - This is a counter of how of this kind of item the character possess. In this instance it is "1".
- *Base Material* - The base material the weapon is compromised of. In this instance the base material is set to "base material".
- *Currency* - The kind of currency needed to purchase the item. It is currently set to "currency".
- *Description* - A simple string description of the item.
- *Masterwork* - A flag as to whether an item is masterwork or not.
- *Item Name* - The main key of the item, it's unique name.
- *Price* - The number, in the currency above required to buy the item.
- *Type* - A string to represent the type of item it is.
- *Weight* - A number that represent how many pounds somethign weighs.
- *Feats* - A list of all the feats that the item grants the user. Currently set at "feat1" and "feat2".
- *Notes Vector* - A list of notes for the item. Currently set at "note1" and "note2".
- *Cursed Notes Vector* - A list of cursed notes for the item. Currently set to "cursed_note1" and "cursed_note2"
- *Skills* - a List of all modifiers that affect skills. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affect a characters skills. Currently set at "2mod""15" for the first pair and "1mod" "5" for the second pair.
- *abilities* - a List of all modifiers that affect abilities. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affect a characters skills. Currently set at "4mod""35" for the first pair and "3mod" "25" for the second pair.
- *combat_stats* - a List of all modifiers that affect combat stats. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affect a characters skills. Currently set at "6mod""55" for the first pair and "5mod" "45" for the second pair.
- *Intelligence item_intelligence* - If the item doesn't have any intelligence then there is nothign inbetween the #'s. If the read in function get's a "#" on it's first read it assume no intelligence and moves on to spell list.
- *Alignment* - A string that holds the items alignment. Currently set to "evil".
- *Communication* - A string to hold how the item communicates with the wielder or others. Currently set to "speech".
- *Senses* - A string for what kind of senses the item has. It's currently set to "great".

- *Purpose* - A string holding the reason the item exists. This one is set "to do evil".
- *Languages* - A vector containing a list of all the languages an item knows. This item knows "language1" and "language2".
- *Lesser Powers* - A vector containing a list of the lesser powers of an item along with how many charges of that power it has. Currently set to "less power 1" with "11" and "less2" with "5".
- *Greater Powers* - A vector containing a list of the Greater powers of an item along with how many charges of that power it has. Currently set to "great power 1" with "22".
- *Dedicated Powers* - A vector containing a list of the dedicated powers of an item along with how many charges of that power it has.Currently set to "dedi 1" with "33"
- *Intelligence* - An int holding a value representing the items intelligence. The higher hte intelligence the more langauges it can know. It current has an intelligence of "20".
- *Wisdom* - An int holding the value of how much wisdom the item has. Currently set to "10".
- *Charisma* - An int holding the level of charisma an item has. Currently set to "3".
- *Num Lesser* - An int holding how many lesser powers an item has. This one is set to "2".
- *Num Greater* - An int holding how many lesser powers an item has. This one is set to "1".
- *Ego* - A integer value of the items Ego. Currently set to "3".
- *Intel Cost* - Integer representing the cost of intelligence of the item. Currently set to "4".
- *Read Langauge* - Boolean value indicating if the item can read langauge. Currently set to true("1").
- *Read Magic* - Boolean value indicating if the item can read magic. This one is currently set to false ("0").
- *Spell List* - A vector with a pair holding a string and integer. The string is the name of the spell, the integer is how many charges the item has of that spell. Currently the item has fireball" and "12" charges.

Magical Armor List Class

Magical armor list is an extended version of the armor list. It contains all the functions, the only real changes were to the file output and input functions, along with the constructor and destructor. All magical items are also masterwork.

```
class magical_armor_list
{
    private:
       map<char*,pair<int,magical_armor*>,ltstr >* magical_armorlist;
       friend class map;

    public:
       magical_armor_list();
       ~magical_armor_list();
       pair<int,magical_armor*> operator[] (char* thekey);

       void add_magical_armor(pair<char*,pair<int,magical_armor*> >i_pair);
       bool remove_magical_armor(char* i_name);
       void clear();

       map<char*,pair<int,magical_armor*>,ltstr >::iterator begin();
       map<char*,pair<int,magical_armor*>,ltstr >::iterator end();

       void write_to_file(char* filename);
       bool read_from_file(char* filename);

       void output();
};
```

Private Member :-

- *map<char*,pair<int,magical_armor*>,ltstr >* magical_armor_list* - A pointer to the actual map that contains magical armors. Each map entry contains a key which is the armor name, the data the map holds is a pair. The pair comprises of an integer representing how many of each item there is, then the second part of the pair is the item instance itself. The string name in the key is actaully a pointer to the item name within the item class. ltstr is a compare function needed by the map to sort the keys. Without it map would be sorting the list by pointer value, not the actual string. A pointer is used as a item list could get rather large, so instead of passing a copy of the whole map by reference a pointer is used to reference the map.

Public Members :-

- *magical_armor_list()* - A constructor that creates the map instance.
- *~magical_armor_list()* - A destructor that deallocates the memory for the item in the pair. Then clears the map and deletes it.
- *pair<int,magical_armor*> operator[] (char* thekey)*
  Overloaded index operator, it will return a copy of the data inthe map that has
the
  key.
- *void add_magical_armor(pair<char*,pair<int,magical_armor*> > i_pair)*

This is a function to add a pair to the map. The problem of masterworkd is not an issue with magical items as all magical items have to be masterwork.So add_magical_item just checks if it exists, if it does it increments it.

- *bool remove_*magical*_armor(char* i_name)*
  This is a simple remove function, a name of an item is sent, if it can be found then it is deleted and a true value is returned. Else a false value is returned if the item cannot be found.
- *void clear()*
  This is a function that can remove all items in the map, it de-allocated all dynamic memory then clears the map.
- *map<char*,pair<int,magical_armor*>,ltstr >::iterator begin()*
  This is a simple function that returns an iterator to the start of the map for parsing through all map values
- *map<char*,pair<int,magical_armor*>,ltstr >::iterator end()*
  This is also a simple function that returns an iterator to the end of the map
- *void write_to_file(char* filename)*
  This function writes all data currently stored in the map to a file by the name filename. It will overwrite any file currently by that name. There are no varying size variables, therefore it simple ordered write to file.
- *bool read_from_file(char* filename)*
  This function reads in data from a text file filename into the map, if file opening fails it will return false. If the file was opened and the function reads to end of file it will return true. All the values are read in, converted to numerical value is neccessary then added to instances of item class. That is then inserted into a pair, which is passed to the add_item function.
- *void output()*
  A test function, this outputs all data currently entered into the map.

File Storage

Magical armor contains 2 classes and multiple vectors and char_int_pairs. This makes storage more complicated as any item can have an infinite number of spells or none. It can be intelligent or not and also each vector coudl be of tremendous size. To define fields a hash("#") was used a segment delimiter. If the item possesses intelligence it also has to store multiple vectors for the langauges it knows etc.

File Storage Structure

Tally Integer
Base Material String
Currency String
Description String
Master Work Bool Value
Item Name String
Price Integer
Type String
Weight Integer
Armor Type String
Armor Size String
Armor Bonus Integer
Armor Max Dexterity Integer
Armor Check Penalty Integer
Armor Arcane Failure Integer
Armor Speed 20 Integer
Armor Speed 30 Integer
Feats Vector<char*>
Notes Vector<char*>
Cursed Notes Vector<char*>
Skills Char_int_pair
abilities Char_int_pair
combat_stats Char_int_pair
Intelligence item_intelligence:-

- Alignment String
- Communication String
- Senses String
- Purpose String
- Languages Vector<char*>
- Lesser Powers Vector<pair<char*,long> >
- Greater Powers Vector<pair<char*,long> >
- Dedicated Powers Vector<pair<char*,long> >
- Intelligence Integer
- Wisdom Integer
- Charisma Integer
- Num Lesser Integer
- Num Greater Integer
- Ego Integer
- Intel Cost Long
- Read Langauge Boolean
- Read Magic Boolean

Spell List Vector<pair<char*,int> >

The orders is:

- *Tally* - This is a counter of how of this kind of item the character possess. In this instance it is "1".
- *Base Material* - The base material the weapon is compromised of. In this instance the base material is set to "base material".
- *Currency* - The kind of currency needed to purchase the item. It is currently set to "currency".
- *Description* - A simple string description of the item.
- *Masterwork* - A flag as to whether an item is masterwork or not.
- *Item Name* - The main key of the item, it's unique name.
- *Price* - The number, in the currency above required to buy the item.
- *Type* - A string to represent the type of item it is.
- *Weight* - A number that represent how many pounds somethign weighs.
- *Armor Type* - A string describing what type of armor it is. Currently set at "Medium".
- *Armor Size* - Another String holding what size categorie the armor fits into. Currently set at "large".
- *Armor Bonus* - An int holding how much of a bonus the armor gives to armor class. Currently set at "4".
- *Armor Max Dexterity* - An integer storing what the max dexterity bonus an armor allows to armor class. Currently set at "3".
- *Armor Check Penalty* - Another integer Storing how much of a penalty certain skills take when donning this armor. Current value is "3".
- *Armor Arcane Failure* - An Int holding the percentage chance a spell will fail. Currently set at "25" %.
- *Armor Speed 20 Integer* - The speed a character can move who can move 20 feet unemcumbered when wearing this armor. Currently set at "15" feet.
- *Armor Speed 30 Integer* - The speed a character can move who can move 30 feet unemcumbered when wearing this armor. Currently set at "20" feet.
- *Feats* - A list of all the feats that the item grants the user. Currently set at "feat1" and "feat2".
- *Notes Vector* - A list of notes for the item. Currently set at "note1" and "note2".
- *Cursed Notes Vector* - A list of cursed notes for the item. Currently set to "cursed_note1" and "cursed_note2"
- *Skills* - a List of all modifiers that affect skills. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affect a characters skills. Currently set at "2mod""15" for the first pair and "1mod" "5" for the second pair.

- *abilities* - a List of all modifiers that affect abilities. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affect a characters skills. Currently set at "4mod""35" for the first pair and "3mod" "25" for the second pair.
- *combat_stats* - a List of all modifiers that affect combat stats. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affect a characters skills. Currently set at "6mod""55" for the first pair and "5mod" "45" for the second pair.
- *Intelligence item_intelligence* - If the item doesn't have any intelligence then there is nothign inbetween the #'s. If the read in function get's a "#" on it's first read it assume no intelligence and moves on to spell list.
- *Alignment* - A string that holds the items alignment. Currently set to "evil".
- *Communication* - A string to hold how the item communicates with the wielder or others. Currently set to "speech".
- *Senses* - A string for what kind of senses the item has. It's currently set to "great".
- *Purpose* - A string holding the reason the item exists. This one is set "to do evil".
- *Languages* - A vector containing a list of all the languages an item knows. This item knows "language1" and "language2".
- *Lesser Powers* - A vector containing a list of the lesser powers of an item along with how many charges of that power it has. Currently set to "less power 1" with "11" and "less2" with "5".
- *Greater Powers* - A vector containing a list of the Greater powers of an item along with how many charges of that power it has. Currently set to "great power 1" with "22".
- *Dedicated Powers* - A vector containing a list of the dedicated of an item along with how many charges of that power it has.Currently set to "dedi 1" with "33"
- *Intelligence* - An int holding a value representing the items intelligence. The higher hte intelligence the more langauges it can know. It current has an intelligence of "20".
- *Wisdom* - An int holding the value of how much wisdom the item has. Currently set to "10".
- *Charisma* - An int holding the level of charisma an item has. Currently set to "3".
- *Num Lesser* - An int holding how many lesser powers an item has. This one is set to "2".
- *Num Greater* - An int holding how many lesser powers an item has. This one is set to "1".
- *Ego* - A integer value of the items Ego. Currently set to "3".
- *Intel Cost* - Integer representing the cost of intelligence of the item. Currently set to "4".

- *Read Langauge* - Boolean value indicating if the item can read langauge. Currently set to true("1").
- *Read Magic* - Boolean value indicating if the item can read magic. This one is currently set to false ("0").
- *Spell List* - A vector with a pair holding a string and integer. The string is the name of the spell, the integer is how many charges the item has of that spell. Currently the item has "fireball" and "12" charges.

Magical Weapon List Class

Magical weapon list is an extended version of the weapon list. It contains all the functions, the only real changes were to the file output and input functions, along with the constructor and destructor. All magical items are also masterwork.

```
class magical_weapon_list
{
   private:
      map<char*,pair<int,magical_weapon*>,ltstr >* magical_weaponlist;
      friend class map;

   public:
      magical_weapon_list();
      ~magical_weapon_list();
      pair<int,magical_weapon*> operator[] (char* thekey);

      void add_magical_weapon(pair<char*,pair<int,magical_weapon*> > i_pair);
      bool remove_magical_weapon(char* i_name);
      void clear();

      map<char*,pair<int,magical_weapon*>,ltstr >::iterator begin();
      map<char*,pair<int,magical_weapon*>,ltstr >::iterator end();

      void write_to_file(char* filename);
      bool read_from_file(char* filename);

      void output();
};
```

Private Member :-

- *map<char*,pair<int,magical_weapon*>,ltstr >* magical_weapon_list* - A pointer to the actual map that contains magical weapons. Each map entry contains a key which is the weapon name, the data the map holds is a pair. The pair comprises of an integer representing how many of each item there is, then the second part of the pair is the item instance itself. The string name in the key is actaully a pointer to the item name within the item class. ltstr is a compare function needed by the map to sort the keys. Without it map would be sorting the list by pointer value, not the actual string. A pointer is used as a item list could get rather large, so instead of passing a copy of the whole map by reference a pointer is used to reference the map.

Public Members :-

- *magical_weapon_list()* - A constructor that creates the map instance.
- *~magical_weapon_list()* - A destructor that deallocates the memory for the item in the pair. Then clears the map and deletes it.
- *pair<int,magical_weapon*> operator[] (char* thekey)*
  Overloaded index operator, it will return a copy of the data in the map that has the key.
- *void add_magical_armor(pair<char*,pair<int,magical_weapon*> > i_pair)*
  This is a function to add a pair to the map. The problem of masterworkd is not an issue with magical items as all magical items have to be masterwork. So add_magical_item just checks if it exists, if it does it increments it.

- *bool remove_magical_weapon(char* i_name)*
  This is a simple remove function, a name of an item is sent, if it can be found then it is deleted and a true value is returned. Else a false value is returned if the item cannot be found.

- *void clear()*
  This is a function that can remove all items in the map, it de-allocated all dynamic memory then clears the map.

- *map<char*,pair<int,magical_weapon*>,ltstr >::iterator begin()*
  This is a simple function that returns an iterator to the start of the map for parsing through all map values

- *map<char*,pair<int,magical_weapon*>,ltstr >::iterator end()*
  This is also a simple function that returns an iterator to the end of the map

- *void write_to_file(char* filename)*
  This function writes all data currently stored in the map to a file by the name filename. It will overwrite any file currently by that name. There are no varying size variables, therefore it simple ordered write to file.

- *bool read_from_file(char* filename)*
  This function reads in data from a text file filename into the map, if file opening fails it will return false. If the file was opened and the function reads to end of file it will return true. All the values are read in, converted to numerical value is neccessary then added to instances of item class. That is then inserted into a pair, which is passed to the add_item function.

- *void output()*
  A test function, this outputs all data currently entered into the map.

File Storage Structure

Magical armor contains 2 classes and multiple vectors and char_int_pairs. This makes storage more complicated as any item can have an infinite number of spells or none. It can be intelligent or not and also each vector coudl be of tremendous size. To define fields a hash("#") was used a segment delimiter. If the item possesses intelligence it also has to store multiple vectors for the langauges it knows etc.

File Storage Example

Tally Integer
Base Material String
CurrencyString
Description String
Master Work Bool Value
Item Name String
Price Integer
Type String
Weight Integer
Weapon Critical String
Weapon Damage String
Weapon Type String
Weapon Handedness String
Weapon Size String
Weapon Rank Integer
Weapon Range Integer
Weapon Two Handed Bonus Bool
Weapon Strength Bonus Bool
Feats Vector<char*>
Notes Vector<char*>
Cursed Notes Vector<char*>
Skills Char_int_pair
abilities Char_int_pair
combat_stats Char_int_pair
Intelligence item_intelligence:-

- Alignment String
- Communication String
- Senses String
- Purpose String
- Languages Vector<char*>
- Lesser Powers Vector<pair<char*,long> >
- Greater Powers Vector<pair<char*,long> >
- Dedicated Powers Vector<pair<char*,long> >
- Intelligence Integer
- Wisdom Integer
- Charisma Integer
- Num Lesser Integer
- Num Greater Integer
- Ego Integer
- Intel Cost Long
- Read Language Boolean
- Read Magic Boolean
- Spell List Vector<pair<char*,int> >

The orders is:

- *Tally* - This is a counter of how of this kind of item the character possess. In this instance it is "1".
- *Base Material* - The base material the weapon is compromised of. In this instance the base material is set to "base material".
- *Currency* - The kind of currency needed to purchase the item. It is currently set to "currency".
- *Description* - A simple string description of the item.
- *Masterwork* - A flag as to whether an item is masterwork or not.
- *Item Name* - The main key of the item, its unique name.
- *Price* - The number, in the currency above required to buy the item.
- *Type* - A string to represent the type of item it is.
- *Weight* - A number that represent how many pounds something weighs.
- *Weapon Critical* - A string representing the number of extra die rolls one gets when they score a critical. Currently Set at "2X".
- *Weapon Damage* - A string holding what kind of damage the weapon does. This one is currently set at "Slashing".
- *Weapon Type* - String with what kind of weapon it is. This one is set to "Light Melee Weapon".
- *Weapon Handedness* - A string holding how the weapon is handled. This one is current set to "One handed".
- *Weapon Size* - A string of the size categories of the weapon. This one is set to "small".
- *Weapon Rank* - An integer for the rank of the weapon. Currently set to "5".
- *Weapon Range* - Int of the maximum range of a weapon before taking a penalty. Currently set at "0".
- *Weapon Two Handed Bonus* - Boolean of whether the item gives a bonus when wielded with both hands. Currently set to false ("0").
- *Weapon Strength Bonus* - A bool of whether a characters strength bonus applies to this weapon. Currently set to True ("1").
- *Feats* - A list of all the feats that the item grants the user. Currently set at "feat1" and "feat2".
- *Notes Vector* - A list of notes for the item. Currently set at "note1" and "note2".
- *Cursed Notes Vector* - A list of cursed notes for the item. Currently set to "cursed_note1" and "cursed_note2"
- *Skills* - a List of all modifiers that affect skills. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of

how much that modifier affects a characters skills. Currently set at "2mod""15" for the first pair and "1mod" "5" for the second pair.

- *abilities* - a List of all modifiers that affect abilities. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affects a characters skills. Currently set at "4mod""35" for the first pair and "3mod" "25" for the second pair.
- *combat_stats* - a List of all modifiers that affect combat stats. It's a pair variable, the first string entry is the name of the modifier. The second value is an integer of how much that modifier affects a characters skills. Currently set at "6mod""55" for the first pair and "5mod" "45" for the second pair.
- *Intelligence item_intelligence* - If the item doesn't have any intelligence then there is nothing in-between the #'s. If the read in function get's a "#" on it's first read it assume no intelligence and moves on to spell list.
- *Alignment* - A string that holds the items alignment. Currently set to "evil".
- *Communication* - A string to hold how the item communicates with the wielder or others. Currently set to "speech".
- *Senses* - A string for what kind of senses the item has. It's currently set to "great".
- *Purpose* - A string holding the reason the item exists. This one is set "to do evil".
- *Languages* - A vector containing a list of all the languages an item knows. This item knows "language1" and "language2".
- *Lesser Powers* - A vector containing a list of the lesser powers of an item along with how many charges of that power it has. Currently set to "less power 1" with "11" and "less2" with "5".
- *Greater Powers* - A vector containing a list of the Greater powers of an item along with how many charges of that power it has. Currently set to "great power 1" with "22".
- *Dedicated Powers* - A vector containing a list of the dedicated powers of an item along with how many charges of that power it has. Currently set to "dedi 1" with "33"
- *Intelligence* - An int holding a value representing the items intelligence. The higher the intelligence the more languages it can know. It current has an intelligence of "20".
- *Wisdom* - An int holding the value of how much wisdom the item has. Currently set to "10".
- *Charisma* - An int holding the level of charisma an item has. Currently set to "3".
- *Num Lesser* - An int holding how many lesser powers an item has. This one is set to "2".
- *Num Greater* - An int holding how many lesser powers an item has. This one is set to "1".

- *Ego* - A integer value of the items Ego. Currently set to "3".
- *Intel Cost* - Integer representing the cost of intelligence of the item. Currently set to "4".
- *Read Language* - Boolean value indicating if the item can read language. Currently set to true("1").
- *Read Magic* - Boolean value indicating if the item can read magic. This one is currently set to false ("0").
- *Spell List* - A vector with a pair holding a string and integer. The string is the name of the spell, the integer is how many charges the item has of that spell. Currently the item has "fireball" and "12" charges.

Item Class

This class was designed to hold the most core aspects of each item. Every item, whether it be magical, a piece of armor or a weapon will have these attributes. A player can have many different items so there can be many instances of this class at once.

```
class item
{
    public:
        item();
        ~item();
        item(const item &new_value);
        item& operator= (const item new_value);
        char* get_item_name();
        bool get_item_masterwork();
        .
        .
        void set_item_name(char* i_name);
        .
        .
        void set_item_masterwork(bool i_masterwork);

    protected:
        char* item_name;
        char* item_currency;
        char* item_description;
        char* item_base_material;
        char* item_type;
        int item_weight;
        int item_price;
        bool item_masterwork;
};
```

Protected Members:

These items are protected because they are needed to be inheritented to all the other item classes.

- *item_name* - Item name is a unique identifier to each item, it's a string and it's used later as a key to store lists of items.
- *item_currency* - The Type of money that is needed to buy the item. E.g copper pieces, gold pieces.
- *item_description* - A simple description of the item along with a description of any special options.
- *item_base_materials* - A description of the basic materials the item was made from.
- *item_type* - A simple string to store the type of item we're storing
- *item_weight* - An integer to store a value relating to how many pounds the item weighs

- *item_price* - An integer the amount of the previously described currency required to purchase the item
- *item_masterwork* - A boolean value to depict whether an item is of high quality. All magical items are masterwork.

<u>Public Members:-</u>

- *item()* - A constructor that sets the string pointers to NULL. This makes it easy to verify whether memory has been allocated for them.
- *~item()* - The dynamic strings need the memory they use to be deallocated. This function merely checks if they're NULL, if not delete's the pointers
- *item(const item &new_value)* - A copy constructor, as the class uses dynamic memory it's important to use a logical copy
- *item& operator*= (const item new_value) - operator= was overloaded just to make workign with items easier, it allocates a new set of memory and copies across all the strings and other values
- *char* get_item_name()*
  char get_item_currency()*
  char get_item_description()*
  char get_item_base_material()*
  char get_item_type()*
  Simple get functions, however they only return the pointer within the class, not a new set of memory.
- *int get_item_weight()*
  int get_item_price()
  Simple return functions that return the value of price and weight.
- *bool get_item_masterwork()*
  Another simple return function that returns a boolen value of whether the item is masterwork
- *void set_item_name(char* i_name)*
  void set_item_currency(char i_currency)*
  void set_item_description(char i_description)*
  void set_item_base_material(char i_base_material)*
  void set_item_type(char i_type)*
  For setting the strings, a character pointer must be passed that points to a string. New memory is allocated and that string is copied over
- *void set_item_weight(int i_weight)*
  void set_item_price(int i_price)
  Pass by value set functions that merely copy the weight and price in.
- *void set_item_masterwork(bool i_masterwork)*
  Pass by value function to set whether an item is masterwork or not.

Armor Class

The armor class is designed to hold all the details that all armors have. It inherits all the private members from item class.

Protected Members:

These members need to be protected as magical_armor class inherits from armor.

- *char* armor_type* - A string to hold what kind of armor it is. e.g Heavy or Light.
- *char* armor_size* - A string that stores how large the armor is, this depicts who can use it, the weight and the cost.
- *int armor_bonus* - An integer to store how much bonus is added to a characters armor class when wearing this armor.
- *int armor_max_dexterity* - An integer which determines the maximum increase to armor class that a characters dexterity can add while wearing this armor.
- *int armor_check_penalty* - Another integer to store how much a penalty a character takes with certain skills when wearing this armor. e.g heavy armor will make it much more difficult to swim compared to leather armor.
- *int armor_arcane_failure* - Again, an integer to store the percent at which u can fail at casting a spell when wearing this armor.
- *int armor_speed_20* - An integer to represent movement speed, some races can move at 20ft when not wearing armor and are slowed with heavy.
- *int armor_speed_30* - Yet another integer to represent movement speed, this one pertains to races that move 30ft unencumbered.

Public Members:-

- *armor()* - A constructor to set all the pointers NULL, it also sets all the inherited pointers to NULL.
- *~armor()* - The destructor which checks all pointers to see if they're NULL, if not it deletes them.
- *char* get_armor_type()*
  char get_armor_size()*
  Simple get function that returns a pointer to the data in the   class.
- *int get_armor_bonus()*
  int get_armor_max_dexterity()
  int get_armor_check_penalty()
  int get_armor_arcane_failure()
  int get_armor_speed_20()
  int get_armor_speed_30()
  Get functions that return a copy of the values in the class instance.

- *void set_armor_type(char* a_armor_type)*
  void set_armor_size(char a_armor_size);*
  Set functions that take in a string, allocate memory the size of the string and copy the data in. The passed string can then be deleted.

- *void set_armor_bonus(int a_armor_bonus)*
  void set_armor_max_dexterity(int a_armor_max_dexterity)
  void set_armor_check_penalty(int a_armor_check_penalty)
  void set_armor_arcane_failure(int a_armor_arcane_failure)
  void set_armor_speed_20(int a_armor_speed_20)
  void set_armor_speed_30(int a_armor_speed_30)
  Very simple pass by reference function that copy the value passed   to the class instances appropriate variable.

Weapon Class

The weapon class is designed to hold all base data pertaining to weapons; it inherits all private members from item. Its private members are protected due to being inherited later by magical_weapon class.

<u>Protected Members:</u>

All members are protected because they are inherited by magical_armor class.

- *char* weapon_critical* - A string to hold the critical multiplier. Due to critical not directly affecting anything it's easier just to store it as a variable with "4x" then as an integer and display a "X" everytime it's outputted. Criticals double to quadruple weapon damage depending on the weapon.
- *char* weapon_damage* - A string which holds the damage a weapon does, it's a record of what die to roll and how many times e.g "2D4" is roll 2, 4 sided die.
- *char* weapon_type* - A string to hold the type of weapon, weapons are classified by the type of damage they do. e.g Blunt.
- *char* weapon_handedness* - A string to hold whether a weapon is light, one handed or two handed.
- *char* weapon_size* - A string to hold what size categorie a weapon fits in. It shows what size creature the weapon was designed for int weapon_rank - A integer to hold the weapon rank.
- *int weapon_range* - An integer to hold the maximum distance the weapon can be throw/shot without takign a penalty
- *bool weapon_two_handed_bonus* - A boolen to hold whether a weapon gives a two handed bonus
- *bool weapon_strength_bonus* - A bool which tell if a weapon gets a bonus from the wielders strength modifier.

<u>Public Members:</u>

- *weapon()* - A constructor to set all the pointers NULL, it also sets all the inherited pointers to NULL.
- *~weapon()* - The destructor which checks all pointers to see if they're NULL, if not it deletes them.
- *char* get_weapon_critical()*
  char get_weapon_damage()*
  char get_weapon_type()*
  char get_weapon_handedness()*
  char get_weapon_size()*
  All simple get functions that return a pointer to the data in the class.
- *int get_weapon_rank()*
  int get_weapon_range()
  Simple function that return a copy of the integer in the class.

- *bool get_weapon_two_handed_bonus()*
  bool get_weapon_strength_bonus()
  Simple functions that return a copy of the boolean in the class.

- *void set_weapon_critical(char* w_weapon_critical)*
  void set_weapon_damage(char w_weapon_damage)*
  void set_weapon_type(char w_weapon_type)*
  void set_weapon_handedness(char w_weapon_handedness)*
  void set_weapon_size(char w_weapon_size)*
  Function that take in a string of characters, calculate the size, allocated memory and copy the data into the class member.

- *void set_weapon_rank(int w_weapon_rank)*
  void set_weapon_range(int w_weapon_range)
  Set function that copies the data passed to it onto the class member.

- *void set_weapon_two_handed_bonus(bool wt_weapon_two_handed_bonus)*
  void set_weapon_strength_bonus(bool w_weapon_strength_bonus)
  A Set Function that copies the boolean value to the class member.

Intelligent Item Class

This is a class that holds all the details for an intelligent magical item. Rare things to come by, it's essentially a mini character a player character can yield and communicate with in different ways. This class stores all info pertaining to them.

Private Members

- *char* alignment* - This is a string variable to hold the characters alignment, whether they're evil,good etc.
- *char* communication* - A string to describe how the item can communicate with a player character. Be it speech, telepathy etc.
- *char* senses* - A string to describe how the item senses events in the world.
- *char* purpose* - A string to describe what purpose the item has to be in the world and what they are tryign to accomplish.
- *vector<char*> languages* - This is a vector that holds strings. An intelligent item can know many languages, Vector being the fast STL sequence container made it a good choice to store all these languages.
- *vector<pair<char*,long> > lesser_powers* - This is a list of all the lesser powers the item possesses. The char* is the name of the lesser power and the long stores how many times it can be used. A pair was used to keep cohesion between the power and how many times that power can be used.
- *vector<pair<char*,long> > greater_powers* - A list of the greater powers an items possess along with how many charges of each it has A pair was used to keep cohesion between the power and how many times that power can be used.
- *vector<pair<char*,long> > dedicated_powers* - The dedicated powers an items possesses and how many charges it has. A pair was used to keep cohesion between the power and how many times that power can be used.
- *int intelligence* - An integer to hold the intelligence the item has, the higher the intelligences the more languages it can know.
- *int wisdom* - A integer that holds the value of wisdom the intelligent item has.
- *int charisma* - This is an integer which holds a value determining how charismatic an item is.
- *int num_lesser* - An integer which holds how many lesser powers an item has.
- *int num_greater* - A integer holding how many greater powers an item possesses.
- *int ego* - An integer to hold the ego a item has.
- *bool read_language* - A boolean that holds whether an item can read.
- *bool read_magic* - A bool to hold whether an item can read magic.

<u>Public Members :</u>

- *item_intelligence()* - A contructor that Sets all the pointers to NULL.
- *item_intelligence(const item_intelligence &new_value)* - A logical copy constructor to copy the pointers values and allocate new memory.
- *~item_intelligence()* - A destructor that free's the pointers memory and clears the vectors.
- *item_intelligence& operator= (const item_intelligence new_value)* - An overloard operator= so intelligence items can be easily copied and worked with.
- *char* get_alignment()*
  char get_communication()*
  char get_senses()*
  char get_purpose()*
  Simple get Functions that return pointers to class strings.
- *vector<char*> get_languages()*
  A function that returns a copy of the vector that holds all the langauges.
- *vector<pair<char*,long> > get_lesser_powers()*
  vector<pair<char,long> > get_greater_powers()*
  vector<pair<char,long> > get_dedicated_powers()*
  Functions that return copies of vectors which hold the items powers and how many of those there are.
- *int get_intelligence()*
  int get_wisdom()
  int get_charisma()
  int get_num_lesser()
  int get_num_greater()
  int get_ego()
  Simple functions that returns copies of class items values.
- *long get_intel_cost()*
  A function that returns a copy of the value of intel cost.
- *bool get_read_language()*
  bool get_read_magic()
  Function that return copies of booleans values indicating whether an item can read languages or magic.
- *void set_alignment(char* i_alignment)*
  void set_communication(char i_communication)*
  void set_senses(char i_senses)*
  void set_purpose(char i_purpose)*
  Simple set functions that take in strings, allocate memory and copy those values.
- *void set_languages(vector<char*> i_languages)*
  Function that takes a vector populated with data and copies all those values into a vector within the class.

- *void set_lesser_powers(vector<pair<char*,long> > i_lesser_powers)*
  void set_greater_powers(vector<pair<char,long> > i_greater_powers)*
  void set_dedicated_power(vector<pair<char,long> > i_dedicated_powers)*
  Functions that take in vectors with pairs of strings and long's. They copy the values of the passed vectors.

- *void set_intelligence(int i_intelligence)*
  void set_wisdom(int i_wisdom)
  void set_charisma(int i_charisma)
  void set_num_lesser(int i_num_lesser)
  void set_num_greater(int i_num_greater)
  void set_ego(int i_ego)
  Simple set function that copies across integer values.

- *void set_intel_cost(long i_intel_cost)*
  Very simple function that copies the long passed into the function into the class instances member.

- *void set_read_language(bool i_read_language)*
  void set_read_magic(bool i_read_magic)
  Set functions that take in boolean values and assign them to class members.

- *void output()* - A function that outputs to the console the value of all members within the class, it is only used for verification that all members are being stored stored correctly.

Magical Item Class

Magic item class is used to store the details magical items can have. Unlike the other item classes, nothign inherits from it, a magical item can have intelligences or spells, however not all do. This class inherits all protected members from the item class.

Private Members :-

- *vector<char*> feats* - A vector of strings to store any feats the item may grant the wielder, there is no limit so it was necessary to use a vector to store the possible multitude of them.
- *vector<char*> notes* - Here is another vector same as above, any number of notes could be stored for the magic item.
- *vector<char*> cursed_notes* - This is a vector of strings storing the details of any curses the item has.
- *char_int_pair skills* - A char_int_pair datatype used to store modifiers that the weapon grants to the users skills.
- *char_int_pair abilities* - Another char_int_pair used to store the additions to the wield abilities the weapon grants.
- *char_int_pair combat_stats* - Yet Another char_int pair, this one contains all the modifiers the weapon grants to combat statisics.
- *item_intelligence* intelligence* - A intelligence class, used to store all details about an intelligent item or NULL if the item isnt' intelligent.
- *vector<pair<char*,int> > spell_list* - A vector that holds pairs of data pertaining to spells. The string is the name of any spells the item grants, then integer holds how many charges of each spell it gives. Item can give an infinite number of different spells all with different amounts of charges.

Public Members :

- *magical_item()* - A constructor, this one sets all the strings in the item class to NULL along with the intelligence pointer.
- *~magical_item()* - A destructor that free's all the strings and also deletes the item intelligence pointer if it's not NULL.
- *void clear(vector<char*>* remove)* - Deprecated function, this was used to deallocate memory from the char*'s within the vectors.
- *vector<char*> get_feats()*
  vector<char> get_notes()*
  vector<char> get_cursed_notes()*
  Simple get functions that return copies of the vectors within the magical item class.
- *char_int_pair get_skills()*
  char_int_pair get_abilities()
  char_int_pair get_combat_stats()
  Get functions that return copies of the char_int_pairs.

- *item_intelligence* get_intelligence()*
  A get function that returns a pointer to the intelligence class item within magical item.
- *vector<pair<char*,int> > get_spell_list()*
  Simple get function returns a copy of the vector spell_list.
- *void set_feats(vector<char*> m_feats)*
  void set_notes(vector<char> m_notes)*
  void set_cursed_notes(vector<char> m_cursed_notes)*
  These are set function that pass a vector that is then copied to the classes vector instance.
- *void set_skills(char_int_pair m_skills)*
  void set_abilities(char_int_pair m_abilities)
  void set_combat_stats(char_int_pair m_combat_stats)
  More simple set skills that merely copy an instance of char_int_pair usign char_int_pairs overloaded operater=.
- *void set_intelligence(item_intelligence* m_intelligence)*
  This set function takes a pointer and merely copies that, it doesn't allocate new memory and copy the details.
- *void set_spell_list(vector<pair<char*,int> > m_spell_list)*
  A set function that takes a copy of a vector and copies all values of it to a new vector in the class.

Magical Weapon Class

The magical weapon class is used to store all extra aspect that a magical weapon has, it inherits all the values from the weapon class, which inherits all the values from item class. The only difference between magical weapon and magical item is the default constructor and destructor.

<u>Private Members :</u>

- *vector<char*> feats* - A vector of strings to store any feats the item may grant the wielder, there is no limit so it was necessary to use a vector to store the possible multitude of them.
- *vector<char*> notes* - Here is another vector same as above, any number of notes could be stored for the magic item.
- *vector<char*> cursed_notes* - This is a vector of strings storing the details of any curses the item has.
- *char_int_pair skills* - A char_int_pair datatype used to store modifiers that the weapon grants to the users skills.
- *char_int_pair abilities* - Another char_int_pair used to store the additions to the wield abilities the weapon grants.
- *char_int_pair combat_stats* - Yet Another char_int pair, this one contains all the modifiers the weapon grants to combat statisics.
- *item_intelligence* intelligence* - A intelligence class, used to store all details about an intelligent item or NULL if the item isnt' intelligent.
- *vector<pair<char*,int> > spell_list* - A vector that holds pairs of data pertaining to spells. The string is the name of any spells the item grants, then integer holds how many charges of each spell it gives. Item can give an infinite number of different spells all with different amounts of charges.

<u>Public Members :-</u>

- *magical_item()* - A constructor, this one sets all the strings in the item class, the weapon class and the intelligence pointer to NULL.
- *~magical_item()* - A destructor that free's all the strings and also deletes the item intelligence pointer if it's not NULL.
- *void clear(vector<char*>* remove)* - Deprecated function, this was used to deallocate memory from the char*'s within the vectors.
- *vector<char*> get_feats()*
  vector<char> get_notes()*
  vector<char> get_cursed_notes()*
  Simple get functions that return copies of the vectors within the magical item class.

- *char_int_pair get_skills()*
  char_int_pair get_abilities()
  char_int_pair get_combat_stats()
  Get functions that return copies of the char_int_pairs.

- *item_intelligence* get_intelligence()*
  A get function that returns a pointer to the intelligence class item within magical item.

- *vector<pair<char*,int> > get_spell_list()*
  Simple get function returns a copy of the vector spell_list.

- *void set_feats(vector<char*> m_feats)*
  void set_notes(vector<char> m_notes)*
  void set_cursed_notes(vector<char> m_cursed_notes)*
  These are set function that pass a vector that is then copied to the classes vector instance.

- *void set_skills(char_int_pair m_skills)*
  void set_abilities(char_int_pair m_abilities)
  void set_combat_stats(char_int_pair m_combat_stats)
  More simple set skills that merely copy an instance of char_int_pair usign char_int_pairs overloaded operater=.

- *void set_intelligence(item_intelligence* m_intelligence)*
  This set function takes a pointer and merely copies that, it doesn't allocate new memory and copy the details.

- *void set_spell_list(vector<pair<char*,int> > m_spell_list)*
  A set function that takes a copy of a vector and copies all values of it to a new vector in the class.

Magical Armor Class

The magical armor class is used to store all extra aspect that a magical armor has, it inherits all the values from the armor class, which inherits all the values from item class. The only difference between magical armor and magical item is the default constructor and destructor.

<u>Private Members :</u>

- *vector<char*> feats* - A vector of strings to store any feats the item may grant the wielder, there is no limit so it was necessary to use a vector to store the possible multitude of them.
- *vector<char*> notes* - Here is another vector same as above, any number of notes could be stored for the magic item.
- *vector<char*> cursed_notes* - This is a vector of strings storing the details of any curses the item has.
- *char_int_pair skills* - A char_int_pair datatype used to store modifiers that the armor grants to the users skills.
- *char_int_pair abilities* - Another char_int_pair used to store the additions to the wield abilities the armor grants.
- *char_int_pair combat_stats* - Yet Another char_int pair, this one contains all the modifiers the armor grants to combat statisics.
- *item_intelligence* intelligence* - A intelligence class, used to store all details about an intelligent item or NULL if the item isnt' intelligent.
- *vector<pair<char*,int> > spell_list* - A vector that holds pairs of data pertaining to spells. The string is the name of any spells the item grants, then integer holds how many charges of each spell it gives. Item can give an infinite number of different spells all with different amounts of charges.

<u>Public Members :</u>

- *magical_item()* - A constructor, this one sets all the strings in the item class, the armor class and the intelligence pointer to NULL.
- *~magical_item()* - A destructor that free's all the strings and also deletes the item intelligence pointer if it's not NULL.
- *void clear(vector<char*>* remove)* - Deprecated function, this was used to deallocate memory from the char*'s within the vectors.
- *vector<char*> get_feats()*
  vector<char> get_notes()*
  vector<char> get_cursed_notes()*
  Simple get functions that return copies of vectors within the magical item class.
- *char_int_pair get_skills()*
  char_int_pair get_abilities()
  char_int_pair get_combat_stats()
  Get functions that return copies of the char_int_pairs.

- *item_intelligence* get_intelligence()*
  A get function that returns a pointer to the intelligence class item within magical item.
- *vector<pair<char*,int> > get_spell_list()*
  Simple get function returns a copy of the vector spell_list.
- *void set_feats(vector<char*> m_feats)*
  void set_notes(vector<char> m_notes)*
  void set_cursed_notes(vector<char> m_cursed_notes)*
  These are set function that pass a vector that is then copied to the classes vector instance.
- *void set_skills(char_int_pair m_skills)*
  void set_abilities(char_int_pair m_abilities)
  void set_combat_stats(char_int_pair m_combat_stats)
  More simple set skills that merely copy an instance of char_int_pair usign char_int_pairs overloaded operater=.
- *void set_intelligence(item_intelligence* m_intelligence)*
  This set function takes a pointer and merely copies that, it doesn't allocate new memory and copy the details.
- *void set_spell_list(vector<pair<char*,int> > m_spell_list)*
  A set function that takes a copy of a vector and copies all values of it to a new vector in the class.

Modifier node

The modifer.h file contains some important structures that are the base components of many of the information classes.

```
struct modifier_node
{
      char* modifier_name;
      int value;
      modifier_node* next;
};
```

The *modifier_node* structure is a generic structure that is the basis for many of the lists within the d20 program. It contains:

- *modifier_name* – This is the name of the element.
- *value* – This is a numeric value of the element.
- *next* – a pointer to the next node.

```
template <class T> struct list_node
{
      T node_value;
      list_node* next;
};
```

This template function takes a template variable an creates a list node for it so that it can be added to a list.

Money

The addition and subtraction of money may be trivial like the previous experience class, but money can take many forms of currency and those currencies may change, as may the difference in value (or exchange rate) between them. For this purpose a class was devised to deal with different forms of currency.

Below is the structure that is manipulated within the money class:

```
struct unit_currency
{
   char* name;
   int value;
   long amount;
};
```

The *unit_currency* structure is used in conjunction with the *money* class to allow varied currencies and exchange rates between the currencies. In original Dungeons and Dragons Rules v3.5 there are four forms of currency, platinum, gold, silver and copper. Each form of currency would have it's own *unit_currency* struct instance. Each variable is explained below:

- *name* – This is used to describe the currency e.g. "gold", "gold pieces" or "gp". It is important that the *name* is kept constant throughout programming as it is the identifying entity of the *unit_currency*. (See money.xml file specification for more information.)
- *value* – Data is structured in such a way that the lowest currency has a base value of 1. Then, each currency above that will contain a *value* that is equal to the number of the base currency it takes to make up one of it's units. (See money.xml file specification for more information.)
- *amount* – This variable holds information on the number of each currency the player has.

The money class stores and manipulates the structure described above:

```
class money
{
   private:
      unit_currency* money_record;
        int size;

   public:
      money();
      money(char*);
      ~money();
      bool read_file(char*);
        bool write_file(char*);

      bool adjust_money(char*, long);
      long operator[] (char*);
      long operator[] (int);
        long get_value (char* money_name);
};
```

<u>Private Members:</u>

- *money_record* – This is a unit_currency pointer (see above for structure description). It stores all the currencies that have been read in from the base file (or character details).
- *size* – This integer variable holds the size (in unit_currencies) of memory allocated to the *unit_currency* pointer.

Public members:

- *money()* - The default constructor sets *money* to be NULL and size to 0.
- *money(char*)* - The initialisation constructor takes a char* parameter which is the name of the file describing each of these currency types. It allocates memory for the *unit_currency* pointer and allocates the size of the pointer to the *size* variable. On error (*bad read* or *file not found*), it notifies the user of an error reading the file.
- *~money()* - destructor is present to clean up any dynamic memory.
- *bool read_file(char*)* – This function takes a char* parameter which is the name of the file describing these currency types external to the constructor. It also allocates memory for the *unit_currency* pointer and allocates size. It returns a bool depending on successful read. It returns *true* for successful read and *false* if an error occurs (*bad read* or *file not found*).
- *bool write_file(char*)* - This function takes a char* parameter which is the name of the file that the function will write to. It writes an xml structure that is described in the money.xml file specification. It will return *true* on a successful write and *false* if an error occurs. **NB. This function will overwrite a file if it already exists.**
- *bool adjust_money(char*, long)* – This function takes a char* parameter which identifies one of the *unit_currency* elements in the *unit_currency* pointer. It also takes a *long* which is the positive or negative adjustment to the number of the specified *unit_currency*. This function returns *true* if the adjustment was successful or *false* if the adjustment could not be made.
- *long operator[] (char*)* – This overloaded operator takes a char* which is the *name* of a *unit_currency* element. It returns the *amount* of the *unit_currency* element if it was found, otherwise, it returns –1.
- *long operator[] (int)* – This overloaded operator takes an int which is the *value* of a *unit_currency* element. It returns the *amount* of the *unit_currency* element if it was found, otherwise, it returns –1.
- *long get_value (char*)* – This function takes a char* which is the *name* of the *unit_currency* element required. It returns the *value* of the *unit_currency* element if it was found, otherwise, it returns –1.

Physical attributes

The physical class was created to store the physical attributes of the character. All of the values are just stored as simple text fields, and no data manipulation is required – the data is used for display purposes only.

```
class physical
{
  private:
    char* char_name;
    char* player_name;
    char* gender;
    char* height;
    char* weight;
    char* hair;
    char* eyes;
    char* skin;
    char* age;
    char* god;
    char* alignment;

  public:
    physical();
    ~physical();

    void set_char_name(char*);
    void set_alignment(char*);
    .
    .
    .
    char* get_char_name();
    char* get_alignment();

    void save_character(char*);
};
```

Private Members:

All private members are char*'s that contain names or descriptions of the new character being generated. The variable names describe the exact information that is stored within each string. (For clarification see the physical file specification that has a small description on each.)

Public Members:

- *physical()* – The default constructor sets all of the char* members of the class to NULL.
- *~physical()* – The destructor deletes all dynamic memory that was generated during the use of the class.

The following is a list of the set functions of the *physical* class. They are named such that the second part of the function name (ie. After set) is the name of the member of the class that will be modified:

void set_char_name(char);*
void set_player_name(char);*
void set_gender(char);*
void set_height(char);*
void set_weight(char);*
void set_hair(char);*
void set_eyes(char);*
void set_skin(char);*
void set_age(char);*
void set_god(char);*
void set_alignment(char);*

The following is a list of the get functions of the *physical* class. They are named such that the second part of the function name (ie. After get) is the name of the member of the class that will be returned:

char get_char_name();*
char get_player_name();*
char get_gender();*
char get_height();*
char get_weight();*
char get_hair();*
char get_eyes();*
char get_skin();*
char get_age();*
char get_god();*
char get_alignment();*

bool read_file(char)* – This function takes a char* parameter which is the name of the file describing the *physical* data. It then populates the *physical* class with the appropriate data. It returns a bool depending on successful read. It returns *true* for successful read and *false* if an error occurs (*bad read* or *file not found*).

bool write_file(char)* - This function takes a char* parameter which is the name of the file that the function will write to. It writes an xml structure that is described in the physical xml file specification. It will return *true* on a successful write and *false* if an error occurs. **NB. This function will overwrite a file if it already exists.**

Physical File Specification:

The file that holds the physical data information is constructed with XML, like so:

```
<d2o>
<physical>
<charName>Arden</charName>
<playerName>Michael</playerName>
<gender>Male</gender>
<height>150</height>
<weight>75</weight>
<hair>Brown</hair>
<eyes>Hazel</eyes>
<skin>Pale white</skin>
<age>20</age>
<god>Obad-hai</god>
<alignment>Neutral</alignment>
</physical>
</d2o>
```

d2o – This tag describes the document bounds.
physical This element contains all that data that will be populated in the physical class.
charName – This element contains the Character's name.
PlayerName – This element conations the player's name.
gender – This element contains the gender of the character.
height – This element contains the height of the character.
weight – This element contains the weight of the character.
hair – This element contains the hair colour of the character.
eyes – This element contains the eye colour of the character.
skin – This element contains the skin colour of the character.
age – This element contains the age of the character.
god – This element contains the god the character worships.
alignment – This element contains the alignment of the character.

profile

The profile class is the class at the center of many of the other classes. It provides an interface between itself and between the classes it contains:

```
class profile
{
      private:
            char_int_group *abilities;
            char_int_group *skills;
            char_int_group *combat;
            physical *physical_prop;
            feat_list *feats;
            feat_list *full_feats;
            Spell_list *spells;
            Spell_list *full_spells;
            xp *experience;
            race *char_race;
            race_selector *race_list;
            race_selector *list_classes;
            class_list* my_classes;
            money *purse;
            int num_skill_points;
            int num_language_points;
            int num_feat_points;
            int num_spell_points;

      public:
            profile();
            ~profile();
            void create_new_character();
            void display_character_stats();
            void create_physical();
            int get_rand_ability_score(int number_rolls);
            int roll_dice(int die);
            void save_character();
            long get_file_length(char*);
            void write_to_file(char* filename, list_node<char*>* &file_list);
            void read_from_file(char* filename);
            void initialise();

            int get_num_skill_points();
            void set_num_skill_points(int point_set);
            int get_num_language_points();
            void set_num_language_points(int point_set);
            int get_num_feat_points();
            void set_num_feat_points(int point_set);
            int get_num_spell_points();
            void set_num_spell_points(int point_set);

            feat_list* get_feats();
            feat_list* get_full_feats();
            Spell_list* get_spells();
            Spell_list* get_full_spells();
            char_int_group* get_abilities();
            char_int_group* get_skills();
            char_int_group* get_combat_stats();
            physical* get_physical();
            xp* get_xp();
            race* get_racial();
            money* get_money();
            race_selector* get_race_list();
            race_selector* get_list_classes();
            class_list* get_my_classes();
};
```

Private Members:

- *char_int_group *abilities* – This char int group contains a list of abilities and their values.
- *char_int_group *skills* – This char int group holds all of the skills that are available.
- *char_int_group *combat* – This char int group holds combat statistics.
- *physical *physical_prop* – This physical instance holds physical data.
- *feat_list *feats* – This feat list holds the feats that the current character has.
- *feat_list *full_feats* – This feat list holds a list of all the feats available.
- *Spell_list *spells* – This spell list holds spells that the character currently has.
- *Spell_list *full_spells* – This spell list holds all the spells that are available.
- *xp *experience* – This xp instance holds experience information.
- *race *char_race* – This race instance hold race information.
- *race_selector *race_list* – This holds a list of races.
- *race_selector *list_classes* – This holds a list of classes.
- *class_list* my_classes* – This holds a list of the characters classes.
- *money *purse* – This variable holds all money information for the character.
- *int num_skill_points* – This variable holds the number of skill points for the character.
- *int num_language_points* – This variable holds the number of language points for the character.
- *int num_feat_points* – This variable holds the number of feat points for the character.
- *int num_spell_points* – This variable holds the number of spell points for the character.

Public Members:

- *profile()* – This constructor calls the initialize function that is described below.
- *~profile()* – Standard Destructor.
- *int get_rand_ability_score(int number_rolls)* – This function takes the number of rolls of a d6 in order to generate a random ability score. The lowest roll is discarded and the total is returned.
- *int roll_dice(int die)* – This function takes a parameter that is the number of sides on a dice and then returns a “random” roll of that dice.
- *void save_character()* – This function saves all of the character data into one file.

- *long get_file_length(char*)* – This function returns the number of bytes that are contained within the file whose name is specified in the first parameter.
- *void write_to_file(char* filename, list_node<char*>* &file_list)* – This function writes to multiple files as specified by the file list and then truncates them into one file with the file name that is the first parameter of this function.
- *void read_from_file(char* filename)* – This function opens the file according to the parameter filename and gets it ready for reading.
- *void initialise()* – The initialize function sets instances of all of the private members if they are pointer and it set integers to 0.
- *int get_num_skill_points()* – This function returns the number of skill points a character has.
- *void set_num_skill_points(int point_set)* – This function takes a parameter and sets the number of skill points according to the parameter.
- *int get_num_language_points()* - – This function returns the number of language points a character has.
- *void set_num_language_points(int point_set)* – This function sets the number of language points according to the parameter.
- *int get_num_feat_points()* - – This function returns the number of feat points a character has.
- *void set_num_feat_points(int point_set)* – This function sets the number of feats points according to the parameter.
- *int get_num_spell_points()* - – This function returns the number of spell points a character has.
- *void set_num_spell_points(int point_set)* – This function sets the number of spell points according to the parameter.
- *feat_list* get_feats()* – This function returns a pointer to the list of feats that a character has.
- *feat_list* get_full_feats()* – This function returns a pointer to the full list of feats.
- Spell_list* get_spells() – *This function returns a pointer to the list of spells the character has.*
- Spell_list* get_full_spells() – *This function returns a pointer to the full list of spells available.*
- *char_int_group* get_abilities()* – This function returns a pointer to a char_int_group of abilities.
- *char_int_group* get_skills()* – This function returns a pointer to a char_int_group of skills.
- *char_int_group* get_combat_stats()* – This function returns a pointer to a char_int_group of combat statistics.

- *physical* get_physical()* – This function returns a pointer to the physical details of the character.
- *xp* get_xp()* – This function returns a pointer to the xp data of the character.
- *race* get_racial()* – This function returns a pointer to the information on the character's race.
- *money* get_money()* – This function returns a pointer to the character's money status.
- *race_selector* get_race_list()* – This function returns a pointer to a race_selector that is able to access race lists.(See race_selector class for more detail.)
- *race_selector* get_list_classes()* – This function returns a pointer to a race_selector that is able to access class lists.
- *class_list* get_my_classes()* – This function returns a pointer to a class_list that contains classes that the character currently has.

Race class

Particular values on a character sheet are affected depending on which *race* was chosen. The numbers that are affected have been collated into a single *race* class so as to provide greater control over them.

```
class race
{
   private:
      char* name;
      char* size;
      int base_speed;
      char** languages;
      char** languages_choice;
      char* fav_class;
      char** special;
      char_int_pair special_mods;
      char* dm_notes;

      int languages_length;
      int languages_choice_length;
      int special_length;

   public:
      race();
      race(char*);
      race(char*,char*,int,char**,char**,char*,char**,
           char_int_pair,char*,int,int,int);
      ~race();

      char* get_name();
      .
      .
      char* get_dm_notes();
      void add_language(char*);

      int get_languages_length();
      int get_languages_choice_length();
      int get_special_length();

      bool read_file(char*, char*);
      bool write_file(char*);
};
```

<u>Private Members:</u>

- *name* - This is a character pointer to the name of the race that the class holds. It is important that this name is kept constant and unique throughout programming as it is the identifying member of this class.
- *size* – This is a character pointer to the descriptive size of the class. For example, “small” or “medium” or “large”.
- *base_speed* – This integer contains the base speed of the race in feet.
- *languages* – This is an array of strings of the languages the class has access to.
- *languages_choice* – This is also an array of strings. It contains languages that are given to the player as bonus languages and it also contains languages that the player chooses.

- *fav_class* – This member contains a char* that holds the race's favoured class.
- *special* – This member contains an array of strings of the modifiers within the race. This is used specifically to retrieve the modifiers from the char_int_pair in the class.
- *special_mods* – This member is a char_int_pair that contains the information on the race modifiers.
- *dm_notes* – This is a char * holding the dmnotes that the race class assigns.

<u>Public members:</u>

- *race()* – This default constructor sets all pointer members to NULL and all integer members to 0.
- *race(char*)* – This first initialisation constructor takes in a char* as it's first parameter. This parameter is the name of the race that will be searched for in the base race file (race.xml). It will then populate the race class with the data it finds. If the base file is not found or if it has been corrupted the race class will not be populated.
- *race(char*,char*,int,char**,char**,char*,char**, char_int_pair, char* ,int, int, int)* – This initialisation constructor information explained below and sets it to the appropriate class members.
    - char*: name of the race.
    - char*: the descriptive size of the race.
    - int: the base speed of the race.
    - char**: an array of language names.
    - char**: an array of language names the player has chosen.
    - char*: the favoured class of the race.
    - char**: a list of the modifier names of the race.
    - char_int_pair: char_int_pair structure of the modifiers.
    - char*: Dm notes for the race.
    - int: length of languages array.
    - int: length of languages_choice array.
    - int: length of special char_int_pair.
- *~race()* – standard destructor is used to clear all dynamic memory.
- *char* get_name()* – This function returns the *name* member of the race class.
- *char* get_size()* – This function returns the *size* member of the race class.
- *int get_base_speed()* – This function returns the *base_speed* member of the race class.
- *char** get_languages()* This function returns the char** member *languages* of the race class.
- *char** get_languages_choice()* – This function returns the char** member *languages_choice* of the race class.
- *char* get_fav_class()* – This function returns the char* *fav_class* member of the race class.

- *char** get_special()* This function returns the char** *special* member of the race class.
- *char_int_pair* get_special_mods()* – This function returns a pointer to the char_int_pair structure *special_mods* in the race class.
- *char* get_dm_notes()* – This function returns the char* *dm_notes* member of the race class.
- *int get_languages_length()* – This function returns the int *languages_length* member of the race class.
- *int get_languages_choice_length()*- This function returns the int *languages_choice* member of the race class.
- *int get_special_length()* – This function returns the int *special_length* member of the race class.
- *void add_language(char*)* – This function adds another language to the languages_choice array.
- *bool read_file(char*, char*)* – The read_file function takes two char*'s. The first is the file name of the file to be read and the second is the race name of the race data to be populated in the race class. The function will return true if it successfully populates the race class. If the file is not found or the file is corrupted, or the race name is not found it will return false.
- *bool write_file(char*)* – This function takes a char* parameter which is the name of the file that the function will write to. It writes an xml structure that is described in the race.xml file specification. It will return *true* on a successful write and *false* if an error occurs. **NB. This function will overwrite a file if it already exists.**

Race File Specification

The file that holds the race data is constructed with XML, like so:

```
<d2o>
<Race>
<Name></Name>
<size></size>
<BaseSpeed></BaseSpeed>
<languages></languages>
<languages_choice></languages_choice>
<FavClass></FavClass>
<Special></Special>
<DmNotes></DmNotes>
</Race>
</d2o>
```

Document Entities

d2o – This element specifies the bounds of the document.

Race – This element holds all the information specific to one race entity.

Name – This element holds the race name that directly relates to the name member in the race class. It should be noted that this is the identifying element of the race entity so it should be kept unique to all other race names.

size – This element holds the descriptive size of the race that directly relates to the *size* member in the race class.

BaseSpeed – This element holds the base speed of the race that directly relates to the base_speed member of the race class.

languages – This element holds the languages the race is allowed and it relates to the *languages* member . Language names are separated by a ':' as seen in the logical document layout below.

languages_choice – This element holds any bonus race languages and languages chosen by the user. It is set up in the same manner as the languages element.

FavClass – This is an element that contains the race's favoured class. It relates directly to the *fav_class* member of the race class.

Special – This holds modifiers for the race. It is set out so the modifier name is first. Then the modifier number is placed in braces '[' ']'. There are no spaces in this string. This layout can be seen below in the Document logical layout.

DmNotes – The DmNotes element hold a string of dm notes.

<u>Document Logical Layout</u>

```
<d2o>
<Race>
<Name>Elf</Name>
<size>medium</size>
<BaseSpeed>30</BaseSpeed>
<languages>elvish:common</languages>
<languages_choice>elvish:common</languages_choice>
<FavClass>Druidic</FavClass>
<Special>onething[-2]anotherthing[3]</Special>
<DmNotes>This is an elf</DmNotes>
</Race>
</d2o>
```

As seen by the layout:

- *languages* and *languages_choice* are separated by a ':'.
- *Special* has a modifier name and then it's modifier in brackets.

Race selector class

The race selector class is a key class a it provides a means of iteration through the race and class files.

```
class race_selector
{
   private:
      list_node<char*>* head;
      int count;

   public:
      race_selector();
      race_selector(char* filename);
      ~race_selector();
      void read_from_file(char* filename);
      char* get_race(int position);
      int get_num_races();
};
```

Private Members:

- *list_node<char*>* head* – This is the head list for the race selector class.
- *int count* – This provides a count on how many or the race or class there are.

Public Members:

- *race_selector()* – This default constructor sets the head member to NULL.
- *race_selector(char* filename)* – This initialisation constructor takes the name of a file as it's parameter and reads that file an populates a list of either race or class names.
- *~race_selector()* – This deconstructor removes all dynamic memory.
- *void read_from_file(char* filename)* – This function takes a file name as it's parameter and proceeds to read the race or class names in from that file.
- *char* get_race(int position)* – This function takes the parameter position and returns the name in the list of the races or class that corresponds to that position in the list.
- *int get_num_races()* – This function returns the number of races or classes contained in the list within this class.

Spell class

There are multiple classes in Dungeons and Dragons, with a very large amount and variety of spells available to choose from. Some spells can be performed by members of different classes, and some are only performable by a particular class and even particular schooling under that class. The *spell* class is designed to hold the important spell-related data together.

```
class Spell
{
  private:
    char* name;
    char* type;
    char* school;
    char_int_pair level;

  public:
    Spell();
    Spell(char*, char*, char*);
    ~Spell();

    void set_name(char*);
    void set_type(char*);
    void set_school(char*);
    void set_next(Spell);

    char* get_name();
    char* get_type();
    char* get_school();

    Spell *next;
};
```

<u>Private members</u>

- *char* name* – This is a character pointer to the name of the spell. It is important that this name is unique, as it is the identifying characteristic of this class.
- *char* type* – This is a character pointer to the spell category this spell falls into, i.e. divine, arcane.
- *char* school* – This is a character pointer to the school in which this spell belongs to, i.e. enchantment, illusion, transmutation.
- *char_int_pair level* - As there are multiple classes that may be able to perform the spell, the *level* requirement is expressed as a *char_int_pair* so as to allow for different level requirements under different classes.

<u>Public members</u>

- *Spell()* – default constructor, sets all char* values to null and also public member pointer *next to null.
- *Spell(char*, char*, char*)* – initialisation constructor, sets all of the char* values to values sent through to the function.
- *~Spell()* – destructor, removes created dynamic memory of all char*'s on exit.
- *void set_name(char*)* – receives a value for the spell name and sets the value to the name private member.

- *void set_type(char*)* – receives a value for the spell type and sets the value to the type private member.
- *void set_school(char*)* – receives a value for the spell school and sets the value to the school private member.
- *char* get_name()* – returns the private member character pointer name
- *char* get_type()* – returns the private member character pointer name
- *char* get_school()* – returns the private member character pointer name
- *Spell *next* – Spell pointer to the next value within the linked list.

Spell list class

This class creates a dynamic linked list of a character's spells.

```
class Spell_list
{
	private:
		Spell* head;
		int count;

	public:
		Spell_list();
		~Spell_list();

		void add_spell(Spell*);
		void remove_spell(char*);

		void clear_list();

		char* operator[](int);
		Spell* get_pointer(int);
		Spell* get_pointer(char*);
		char* get_spell_name(int);
		int get_num_spells();
		Spell* operator() (int);
		Spell* operator() (char*);

		int write_to_file(char* filename);
		int read_from_file(char* filename);
};
```

Private Members:

- *Spell* head* – A pointer to the head spell instance of the spell list.
- *int count* – An integer specifying the number of spells in the list.

Public Members:

- *Spell_list()* - This is the default constructor that sets count to 0 and the spell pointer to NULL.
- *~Spell_list()* – The class destructor that clears the spell list and deletes any dynamic memory.
- *void add_spell(Spell*)* – This takes a pointer to the instance of the spell class and adds it to the spell list.
- *void remove_spell(char*)* – This function accepts a char* that is the name of the spell and removes it from the spell list.
- *void clear_list()* – This function removes all entries from the current spell list.
- *char* operator[](int)* – This operator accepts an int that relates to a position in the spell list (similar to an array index.) It returns the spell *name* of the current instance in the list.
- *Spell* get_pointer(int)* – This function returns a pointer to the Spell class instance that is kept in the position that is specified by the *int* parameter.

- *Spell* get_pointer(char*)* – This function also returns a pointer to the Spell class instance but this time the parameter is the name of the spell pointer to be retrieved.
- *char* get_spell_name(int)* – This function returns the spell name as a char* of the spell instance that is referred to by the first parameter of the function.
- *int get_num_spells()* – This function returns the number of spells that are in the spell class.
- *Spell* operator() (int)* – This operator take a position parameter and returns a pointer to the instance that is contained in that position of the list.
- *Spell* operator() (char*)* – This operator takes a char* that is the name of a spell and returns a pointer to the spell class that corresponds to it. If it is not found NULL will be returned.
- *int write_to_file(char* filename)* – This function takes the first parameter which is the name of the file to be written to and writes all the spell data that is contained in the spell list in a format: *name:type:school:* to the file. It returns 1 on success or 0 on failure.
- *int read_from_file(char* filename)* – This function takes the first parameter which is the name of the file to read and reads all spell data into the spell list in the same format as specified above. It will return 1 on success and 0 on failure.

Utilities

```
#include <fstream>
using namespace std;

void int_to_string(int value, bool show_sign, int num_digits, char* output);
int create_index_array(long* &index_array, char* filename, char seperator);
```

There have been a small number of functions that are generally used through out the d20 program.

- *void int_to_string(int, bool, int, char*)* – This function takes an integer value and expresses it as a string, complete with sign and preceeding 0's if specified.
- *int*: Integer storing the value to be converted into a string
- *bool*: Boolean value specifying whether the sign of the number should be displayed if the number is positive.
- *int*: Integer specifying the number of digits to be displayed. Digits are displayed from the rightmost and extra digits are 0's.
- *char*:* Character array that the string gets written to. There is no checking to see if the string is going to fit.

- *int create_index_array(long**, char*, char)* - This function takes a filename and a separator character, turning it into a long integer array, specifying the beginning of a new line. The number of "lines" found is then returned. It has the following parameters:
- *long*** - A long integer pointer, which is allocated dynamic memory to form an array where all the index values are to be stored. It is passed by reference so that the calling function can use the index after the memory is allocated
- *char** - A character array which holds the name of the file to be indexed.

- *char* - A character that is to be used as the separator. A new "line" is assumed after one of these characters is encountered.

- *char* long_to_string(long)*
  char int_to_string(int) -*
  These functions both perform the same task. The difference is that they accept different parameters. Each of them accept a number (integer or long depending on the function call.) as it's only parameter. It then converts the number to the string equivalent and returns a char* to it.

- *char digit_to_char(int)* – This function is used by the functions. This function accepts an int (that should be a single numeric digit and it returns the corresponding character to that digit.

Experience (XP) class

The experience class is a very simple class which was only intended so as to control any changes to XP from within its own class. This is mainly preventative of changes made from other functions accidentally.

```
class xp
{
      private:
            long experience;

      public:
            xp();
            xp(long);
            ~xp();

            void adjust_xp(long);
            long get_xp();

            bool read_file(char*);
            bool write_file(char*);
};
```

<u>Private Members:</u>
xp - The experience points (XP) are stored as a long integer because it is understood that the actual XP value can rise to quite a large number.

<u>Public Members:</u>
xp() - The default constructor simply initialises xp to a zero value.

xp(long) - the initialisation constructor will obviously initialise the value to the long integer given as a parameter.

void adjust_xp(long) – This function adjusts the value of xp by the given amount. This includes adding to or subtracting from depending on whether the amount is positive or negative.

long get_xp() – This function simply returns the value of xp.

bool read_file(char)* – This function takes a *char** that is the name of the file to read the xp data in from. It will return *true* on success and *false* on fail.

bool write_file(char*) – *This function takes a char* that is the name of the file it will write xp data to. It will return* true *on success and* false *on fail.*

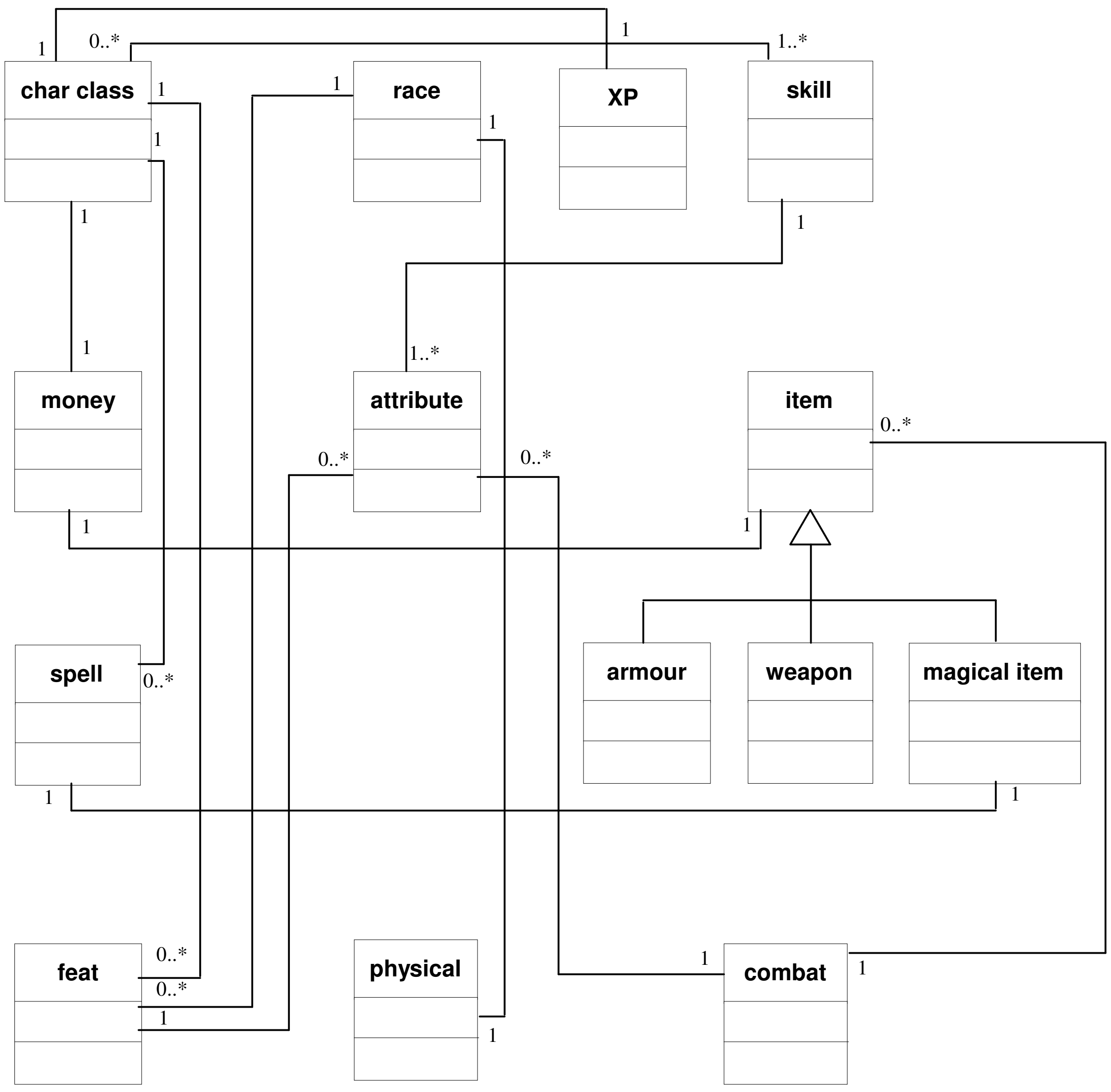
<u>xp File Specification</u>

The xp file is a very simplistic structure.

```
<d2o>
<xp>
<experience>3440</experience>
</xp>
</d2o>
```

d2o – Specifies the bounds of the document.
xp – This contains experience information. (Not extremely necessary at the moment but when multi-classing is implemented it will be required.)
experience – This element contains the number of experience points.

Class diagram and relationships



Links in this diagram show relationships between the classes at a high level. The main purpose for this diagram is to allow easy identification of other classes that may be affected if class information is changed.

This class diagram focuses closely on the implementation of the main classes from the section above. Links in this diagram show relationships between the classes at a high level. The main purpose for this diagram is to allow easy identification of other classes that may be affected if class information is changed. *(For a complete specification of the system as a class diagram see index. This will be added in the next revision of the manual.)*

The Character Environment is a broad concept that encompasses the user interface and the shell.
Particular attention has to be paid to the *item* class as it has three sub-classes:

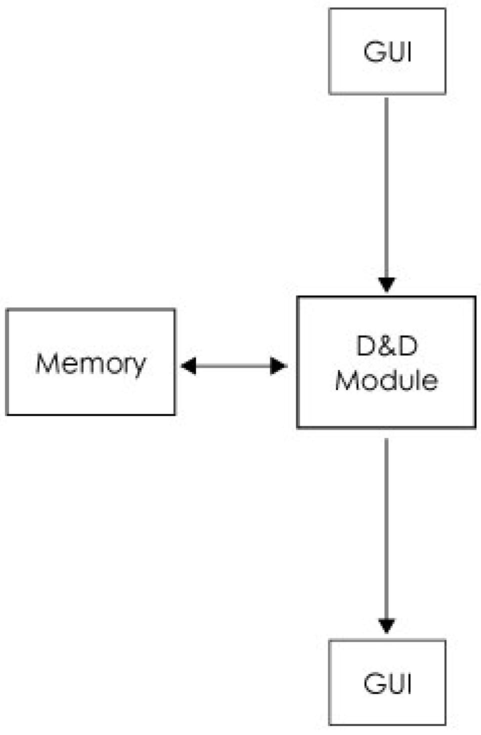
- Weapon
- Armour
- Magic item

The skill class' unary relationship is also of interest. Modification of a skill object can have an effect on other skill objects.

All other relationships are fairly straight-forward.

GUI interaction

When the user interacts with the GUI, i.e. when they add their abilities scores into the system, the GUI sends signals to the D&D module functions that have slots listening for signals. When a slot hears a signal, the function in the D&D module then interacts with the memory setup for the program. It can either sent information, receive information or do both. Once the function has completed it's processing of data it then sends a signal back to the GUI, that in turns makes the appropriate changes to the data displayed to the user.



Naming Standards for GUI variables:

- Labels: label_parent_name
- Buttons: btn_parent _name
- Text Boxes: txtb_parent _name
- Main Menu: mnu_parent _name
- Check Boxes: ckb_parent _name
- Group Boxes: grp_parent _name
- Picture Boxes: picb_parent _name
- Panels: pnl_parent _name
- Data Grids: dtag_parent _name
- List Boxes: lstb_parent _name
- Combo Boxes: comb_parent _name
- List Views: lstv_parent _name

- Tree Views: trev_parent _name
- Tab Controls: tabc_parent _name
- Tab Pages: tabp_parent _name
- Numeric Up Down: nud_name
- Tool Tips: tolt_name
- Open File Dialogs: ofd_name
- Save File Dialogs: sfd_name
- Print Dialogs: prtd_name
- Print Preview Dialogs: prtpd_name
- Print Document: prtdoc_name

GUI breakdown

The GUI is broken up into a number of sections, the application main menu (d2o_starter), the character editor (d20_characters), the magic item creator (d2o_new_items), the house rules editor (d2o_house_rules) and a miscellaneous form for setting proficiencies (d2o_misc).

The way in which .NET works meant that only the primary dialog required any set up in the .cpp file. All other .cpp files ack as a link for the .h file to the appropriate resources.

The following pages will give a detailed explanation of each section of the GUI. There are 5 classes that comprise the GUI dialog.

d2o_starter:

This function initiates the main menu for the GUI and application. d2o_starter.cpp is the only file that has any more then just a link for the .h files. The following it the content of d2o_starter.cpp

```
#include "StdAfx.h"
#include "d2o_starter.h"
#include <windows.h>

using namespace d2o;


int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)

{
      //Starting application thread for the forms/GUI
      System::Threading::Thread::CurrentThread->ApartmentState=
system::Threading::ApartmentState::STA;


      //Starts the d2o_starter.h form
      Application::Run(new d2o_starter());
      return 0;
}
```

`int APIENTRY _tWinMain()` Is .NET's function that initiates the GUI. From here it creates an instance of the d2o_starter class (form) into an awaiting thread.

d2o_starter.h

<u>Include files:</u>

```
#pragma once
#include "d20_house_rules.h"
#include "d2o_characters.h"
#include "d2o_class_maker.h"
```

<u>Namespace:</u>

```
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
```

Public Members :-

This is the class's (form/GUI) constructor. The function goes through and declares an instance an initialises of each of the .NET components that will be used within the GUI:

```
d2o_starter(void)
{
      InitializeComponent();
}
```

Protected Members :-

This is a common de-constructor for .NET forms. It is found in every .h file.

```
void Dispose(Boolean disposing)
{
      if (disposing && components)
      {
            components->Dispose();
      }
      __super::Dispose(disposing);
}
```

Private Member :-

For a more in-depth breakdown of the functionality of the private members, please refer to the in code documentation.

Component declareatations: (Standard .NET protocal)

```
System::Windows::Forms::Button *  btn_new_character;
System::Windows::Forms::Button *  btn_load_caharcter;
System::Windows::Forms::Button *  btn_house_rules;
System::Windows::Forms::MainMenu *  mnu_file;
System::Windows::Forms::MenuItem *  mnui_file_title;
System::Windows::Forms::MenuItem *  mnui_new;
System::Windows::Forms::MenuItem *  mnui_load;
System::Windows::Forms::MenuItem *  mnui_house_rules;
System::Windows::Forms::OpenFileDialog *  ofd_load;
System::Windows::Forms::MenuItem *  mnui_exit;
System::ComponentModel::Container* components;
```

Funtiocn declareatation:

`void InitializeComponent(void):` This function initiates the components used in this form/GUI.Also the function allocates the memory for all the components and there content.

`mnui_new_Click(System::Object * sender, System::EventArgs * e):` This function opens the new character dialog the form.

`mnui_exit_Click(System::Object * sender, System::EventArgs * e):` This function closes the current dialog form.

`mnui_load_Click ( System::Object * sender, System::EventArgs * e):` This function opens the load file dialog form.

`ofd_load_FileOk(System::Object * sender, System::ComponentModel::CancelEventArgs * e):` This function displays users request to load a new file.

`ofd_load_HelpRequest(System::Object * sender, System::EventArgs * e):` This function displays the help box to the user on their request.

d20_characters.h:

This class handles the entire front end GUI interaction of the new character editor.

Include files:

```
#pragma once
#include "d2o_new_items.h"
#include "d2o_misc.h"
```

Namespace:

```
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
```

Public Members :-

This is the class's (form/GUI) constructor. The function goes through and declares an instance an initialises of each of the .NET components that will be used within the GUI:

```
      d2o_characters(void)
      {
            InitializeComponent();
      }
```

Protected Members :-

Same as d2o_starter.h Protected Members.

Private Member :-

Due to the large amount of private members in d20_characters.h only a same will be provided. Please see inline documentation form more information.

Component declareatations: (Standard .NET protocal)

```
      System::Windows::Forms::GroupBox *  grpb_classes;
      System::Windows::Forms::GroupBox *  grpb_level;
      System::Windows::Forms::TextBox *  txtb_level;
      ...
      System::Windows::Forms::TextBox *  txtb_hp_mod;
      System::Windows::Forms::TextBox *  txtb_hp;
      System::ComponentModel::Container* components;
```

Funtiocn declareatation:

`void InitializeComponent(void):` This function initiates the components used in this form/GUI.Also the function allocates the memory for all the components and there content.

Please see d20_characters.h for all function documentation.

d2o_new_items.h:

This class handles the entire front end GUI interaction of the new magic item editor.

Include files:
```
#pragma once
```

Namespace:
```
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
```

Public Members :-

This is the class's (form/GUI) constructor. The function goes through and declares an instance an initialises of each of the .NET components that will be used within the GUI:
```
        d2o_characters(void)
        {
              InitializeComponent();
        }
```

Protected Members :-

Same as d2o_starter.h Protected Members.

Private Member :-

Due to the large amount of private members in d20_characters.h only a same will be provided. Please see inline documentation form more information.

Component declareatations: (Standard .NET protocal)

```
        System::Windows::Forms::Button *  btn_apply_new_item;
        System::Windows::Forms::Button *  btn_cancel_new_item;
        System::Windows::Forms::TabControl *  tabc_new_items;
        ...
        System::Windows::Forms::CheckBox *  ckb_cursed;
        System::Windows::Forms::Label *  label_intelligent_ego;
        System::Windows::Forms::TextBox *  txtb_intelligent_ego;
```

Funtiocn declareatation:

`void InitializeComponent(void)`: This function initiates the components used in this form/GUI.Also the function allocates the memory for all the components and there content.

Please see d20_characters.h for all function documentation.

d2o_misc.h:

This class handles the entire front end GUI interaction of the miscellaneous skill selector.

Include files:
```
#pragma once
```

Namespace:
```
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
```

Public Members :-

This is the class's (form/GUI) constructor. The function goes through and declares an instance an initialises of each of the .NET components that will be used within the GUI:
```
        d2o_characters(void)
        {
              InitializeComponent();
        }
```

Protected Members :-

Same as d2o_starter.h Protected Members.

Private Member :-

Due to the large amount of private members in d20_characters.h only a same will be provided. Please see inline documentation form more information.

Component declareatations: (Standard .NET protocal)

```
        System::Windows::Forms::GroupBox *  grpb_classes;
        System::Windows::Forms::GroupBox *  grpb_level;
        System::Windows::Forms::TextBox *  txtb_level;
        ...
        System::Windows::Forms::TextBox *  txtb_hp_mod;
        System::Windows::Forms::TextBox *  txtb_hp;
        System::ComponentModel::Container* components;
```

Funtiocn declareatation:

`void InitializeComponent(void):` This function initiates the components used in this form/GUI.Also the function allocates the memory for all the components and there content.

`btn_add_Click(System::Object *  sender, System::EventArgs *  e):` This function closes the form if the user click the button or hits enter.

`btn_cancel_Click(System::Object * sender, System::EventArgs * e):` This function closes the form if the user click the button or hits escape.

Please see d20_characters.h for all function documentation.

d2o_house_rules.h:

This class handles the entire front end GUI interaction of the house rules editor.

Include files:
```
#pragma once
```

Namespace:
```
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
```

Public Members :-

This is the class's (form/GUI) constructor. The function goes through and declares an instance an initialises of each of the .NET components that will be used within the GUI:
```
        d2o_characters(void)
        {
              InitializeComponent();
        }
```

Protected Members :-

Same as d2o_starter.h Protected Members.

Private Member :-

Due to the large amount of private members in d20_characters.h only a same will be provided. Please see inline documentation form more information.

Component declareatations: (Standard .NET protocal)

```
        System::Windows::Forms::TabControl *  tabc_house_rules;
        System::Windows::Forms::TabPage *  tabp_items;
        System::Windows::Forms::TabPage *  tabp_spells;
        ...
        System::Windows::Forms::Label *  label_domains_information;
        System::Windows::Forms::ComboBox *  comb_domains_data;
        System::Windows::Forms::NumericUpDown* nud_items_weapons_weight;
```

Funtiocn declareatation:

`void InitializeComponent(void):` This function initiates the components used in this form/GUI.Also the function allocates the memory for all the components and there content.

Please see d20_characters.h for all function documentation.

Important tables

Table of contents:

Table 3.1 – Base Save and Base Attack Bonuses

Class level	Base Save Bonus (Good)	Base Save Bonus (Poor)	Base Attack Bonus (Good)	Base Attack Bonus (Average)	Base Attack Bonus (Poor)
1st	+2	+0	+1	+0	+0
2nd	+3	+0	+2	+1	+1
3rd	+3	+1	+3	+2	+1
4th	+4	+1	+4	+3	+2
5th	+4	+1	+5	+3	+2
6th	+5	+2	+6/+1	+4	+3
7th	+5	+2	+7/+2	+5	+3
8th	+6	+2	8/+3	+6/+1	+4
9th	+6	+3	+9/+4	+6/+1	+4
10th	+7	+3	+10/+5	+7/+2	+5
11th	+7	+3	+11/+6/+1	+8/+3	+5
12th	+8	+4	+12/+7/+2	+9/+4	+6/+1
13th	+8	+4	+13/+8/+3	+9/+4	+6/+1
14th	+9	+4	+14/+9/+4	+10/+5	+7/+2
15th	+9	+5	+15/+10/+5	+11/+6/+1	+7/+2
16th	+10	+5	+16/+11/+6/+1	+12/+7/+2	+8/+3
17th	+10	+5	+17/+12/+7/+2	+12/+7/+2	+8/+3
18th	+11	+6	+18/+13/+8/+3	+13/+8/+3	+9/+4
19th	+11	+6	+19/+14/+9/+4	+14/+9/+4	+9/+4
20th	+12	+6	+20/+15/+10/+5	+15/+10/+5	+10/+5

Table 3.2 – Experience and level-dependent benefits

Ch'ter level	XP	Class Skill Max Ranks	Cross-class Skill Max Ranks	Feats	Ability Score Increases
1st	0	4	2	1st	–
2nd	1,000	5	2-1/2	–	–
3rd	3,000	6	3	2nd	–
4th	6,000	7	3-1/2	–	1st
5th	10,000	8	4	–	–
6th	15,000	9	4-1/2	3rd	–
7th	21,000	10	5	–	–
8th	28,000	11	5-1/2	–	2nd
9th	36,000	12	6	4th	–
10th	45,000	13	6-1/2	–	–
11th	55,000	14	7	–	–
12th	66,000	15	7-1/2	5th	3rd
13th	78,000	16	8	–	–
14th	91,000	17	8-1/2	–	–
15th	105,000	18	9	6th	–
16th	120,000	19	9-1/2	–	4th
17th	136,000	20	10	–	–
18th	153,000	21	10-1/2	7th	–
19th	171,000	22	11	–	–
20th	190,000	23	11-1/2	–	5th

Table 3.7 – Dieties

Deity	Alignment	Domains	Typical Worshipers
Heironeous, god of valor	Lawful good	Good, Law, War	Paladins, fighters, monks
Moradin, god of the dwarves	Lawful good	Earth, Good, Law, Protection	Dwarves
Yondalla, goddess of the halflings	Lawful good	Good, Law, Protection	Halflings
Ehlonna, goddess of the woodlands	Neutral good	Animal, Good, Plant, Sun	Elves, gnomes, half-elves, halflings, rangers, druids
Garl Glittergold, god of the gnomes	Neutral good	Good, Protection, Trickery	Gnomes
Pelor, god of the sun	Neutral good	Good, Healing, Strength, Sun	Rangers, bards
Corellon Larethian, god of the elves	Chaotic good	Chaos, Good, Protection, War	Elves, half-elves, bards
Kord, god of strength	Chaotic good	Chaos, Good, Luck, Strength	Fighters, barbarians, rogues, athletes
Wee Jas, goddess of death and magic	Lawful neutral	Death, Law, Magic	Wizards, necromancers, sorcerers
St. Cuthbert, god of retribution	Lawful neutral	Destruction, Law, Protection, Strength	Fighters, monks, soldiers
Boccob, god of magic	Neutral	Knowledge, Magic, Trickery	Wizards, sorcerers, sages
Fharlanghn, god of roads	Neutral	Luck, Protection, Travel	Bards, adventurers, mercharnts
Obad-Hai, god of nature	Neutral	Air, Animal, Earth, Fire, Plant, Water	Druids, barbarians, rangers
Olidammara, god of thieves	Chaotic neutral	Chaos, Luck, Trickery	Rogues, bards, thieves
Hextor, god of tyranny	Lawful evil	Destruction, Evil, Law, War	Evil fighters, monks
Nerull, god of death	Neutral evil	Death, Evil, Trickery	Evil necromancers, rogues
Vecna, god of secrets	Neutral evil	Evil, Knowledge, Magic	Evil wizards, sorcerers, rogues, spies
Erythnul, god of slaughter	Chaotic evil	Chaos, Evil, Trickery, War	Evil fighter, barbarians, rogues
Gruumsh, god of the orcs	Chaotic evil	Chaos, Evil, Strength, War	Half-orcs, orcs

Table 2.1 – Racial ability adjustments

Race	Ability Adjustment	Favoured Class
Human	None	Any
Dwarf	+2 Constitution, -2 Charisma	Fighter
Elf	+2 Dexterity, -2 Strength	Wizard
Gnome	+2 Constitution, -2 Strength	Bard
Half-elf	None	Any
Half-orc	+2 Strength, -2 Intelligence[1], -2 Charisma	Barbarian
Halfling	+2 Dexterity, -2 Strength	Rogue
1 A half-orc's starting Intelligence score is always at least 3. If this adjustment would lower the character's score to 1 or 2, his score is nevertheless 3.		

Table 6.4 – Random starting ages

Race	Adulthood	Barbarian Rogue Sorcerer	Bard Fighter Paladin Ranger	Cleric Druid Monk Wizard
Human	15 years	+1d4	+1d6	+2d6
Dwarf	40 years	+3d6	+5d6	+7d6
Elf	110 years	+4d6	+6d6	+10d6
Gnome	40 years	+4d6	+6d6	+9d6
Half-elf	20 years	+1d6	+1d6	+2d6
Half-orc	14 years	+1d4	+1d6	+2d6
Halfling	20 years	+2d4	+3d6	+4d6

Table 6.6 – Random height and weight association

Race	Base Height	Height modifier	Base weight	Weight modifier
Human, male	4' 10"	+2d10	120 lb.	X (2d4) lb.
Human, female	4' 5"	+2d10	85 lb.	X (2d4) lb.
Dwarf, male	3' 9"	+2d4	130 lb.	X (2d6) lb.
Dwarf, female	3' 7"	+2d4	100 lb.	X (2d6) lb.
Elf, male	4' 5"	+2d6	85 lb.	X (1d6) lb.
Elf, female	4' 5"	+2d6	80 lb.	X (1d6) lb.
Gnome, male	3' 0"	+2d4	40 lb.	X 1 lb.
Gnome, female	2' 10"	+2d4	35 lb.	X 1 lb.
Half-elf, male	4' 7"	+2d8	100 lb.	X (2d4) lb.
Half-elf, female	4' 5"	+2d8	80 lb.	X (2d4) lb.
Half-orc, male	4' 10"	+2d12	150 lb.	X (2d6) lb.
Half-orc, female	4' 5"	+2d12	110 lb.	X (2d6) lb.
Halfling, male	2' 8"	+2d4	30 lb.	X 1 lb.
Halfling, female	2' 6"	+2d4	25 lb.	X 1 lb.

Table 2: Hit die / class

HD Type	Class
d4	Sorcerer, wizard
d6	Bard, rogue
d8	Cleric, druid, monk, ranger
d10	Fighter, paladin
d12	Barbarian

Table 7.4 – Tiny & large weapon damage

Medium weapon damage	Tiny weapon damage	Large weapon damage
1d2	–	1d3
1d3	1	1d4
1d4	1d2	1d6
1d6	1d3	1d8
1d8	1d4	2d6
1d10	1d6	2d8
1d12	1d8	3d6
2d4	1d4	2d6
2d6	1d8	3d6
2d8	1d10	3d8
2d10	2d6	4d8

Table 7.3 – Trade goods

Cost	Item
1cp	One pound of wheat
2cp	One pound of flour, or one chicken
1sp	One pound of iron
5sp	One pound of tobacco or copper
1gp	One pound of cinnamon, or one goat
2gp	One pound of giner or pepper, or one sheep
3gp	One pig
4gp	One square yard of linen
5gp	One pound of salt or silver
10gp	One square yard of silk, or one cow
15gp	One pound of saffron, or cloves, or one ox
50gp	One pound of gold
500gp	One pound of platinum

Table 4-1: Skill Points Per Level

Class	1st-Level Skill Points[1]	Higher-Level Skill Points[2]
Barbarian	(4 + Int modifier) * 4	4 + Int modifier
Bard	(6 + Int modifier) * 4	6 + Int modifier
Cleric	(2 + Int modifier) * 4	2 + Int modifier
Druid	(4 + Int modifier) * 4	4 + Int modifier
Fighter	(2 + Int modifier) * 4	2 + Int modifier
Monk	(4 + Int modifier) * 4	4 + Int modifier
Paladin	(2 + Int modifier) * 4	2 + Int modifier
Ranger	(6 + Int modifier) * 4	6 + Int modifier
Rogue	(8 + Int modifier) * 4	8 + Int modifier
Sorcerer	(2 + Int modifier) * 4	2 + Int modifier
Wizard	(2 + Int modifier) * 4	2 + Int modifier
1 **Humans** add +4 to this total at 1st-Level.		
2 **Humans** add +1 to each new level.		

Table 7-1: Random Starting Gold

Class	Amount (average)	Class	Amount (average)
Barbarian	4d4 * 10 (100gp)	Paladin	6d4 * 10 (150gp)
Bard	4d4 * 10 (100gp)	Ranger	6d4 * 10 (150gp)
Cleric	5d4 * 10 (125gp)	Rogue	5d4 * 10 (125gp)
Druid	2d4 * 10 (50gp)	Sorcerer	3d4 * 10 (75gp)
Fighter	6d4 * 10 (150gp)	Wizard	3d4 * 10 (75gp)
Monk	5d4 (12gp, 5sp)		

Table 4-2: Skills

Skill	Bbn	Brd	Clr	Drd	Ftr	Mnk	Pal	Rgr	Rog	Sor	Wiz	Untrained	Key Ability
Appraise	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Int
Balance	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	Yes	Dex1
Bluff	cc	C	cc	cc	cc	cc	cc	cc	C	C	cc	Yes	Cha
Climb	C	C	cc	cc	C	C	cc	C	C	cc	cc	Yes	Str1
Concentration	cc	C	C	C	cc	C	C	C	cc	C	C	Yes	Con
Craft	C	C	C	C	C	C	C	C	C	C	C	Yes	Int
Decipher Script	cc	C	cc	cc	cc	cc	cc	cc	C	cc	C	No	Int
Diplomacy	cc	C	C	C	cc	C	C	cc	C	cc	cc	Yes	Cha
Disable Device	cc	cc	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Int
Disguise	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Cha
Escape Artist	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	Yes	Dex1
Forgery	cc	cc	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Int
Gather Information	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Cha
Handle Animal	C	cc	cc	C	C	cc	C	C	cc	cc	cc	No	Cha
Heal	cc	cc	C	C	cc	cc	C	C	cc	cc	cc	Yes	Wis
Hide	cc	C	cc	cc	cc	C	cc	C	C	cc	cc	Yes	Dex1
Intimidate	C	cc	cc	cc	C	cc	cc	cc	C	cc	cc	Yes	Cha
Jump	C	C	cc	cc	C	C	cc	C	C	cc	cc	Yes	Str1
Knowledge (arcana)	cc	C	C	cc	cc	C	cc	cc	cc	C	C	No	Int
Knowledge (architecture and engineering)	cc	C	cc	cc	cc	cc	cc	cc	cc	cc	C	No	Int
Knowledge (dungeoneering)	cc	C	cc	cc	cc	cc	cc	C	cc	cc	C	No	Int
Knowledge (geography)	cc	C	cc	cc	cc	cc	cc	C	cc	cc	C	No	Int
Knowledge (history)	cc	C	C	cc	cc	cc	cc	cc	cc	cc	C	No	Int
Knowledge (local)	cc	C	cc	cc	cc	cc	cc	cc	C	cc	C	No	Int
Knowledge (nature)	cc	C	cc	C	ccc	cc	cc	C	cc	cc	C	No	Int
Knowledge (nobility and royalty)	cc	C	cc	cc	cc	cc	C	cc	cc	cc	C	No	Int
Knowledge (religion)	cc	C	C	cc	cc	C	C	cc	cc	cc	C	No	Int
Knowledge (the planes)	cc	C	C	cc	cc	cc	cc	cc	cc	cc	C	No	Int
Listen	C	C	cc	C	cc	C	cc	C	C	cc	cc	Yes	Wis
Move Silently	cc	C	cc	cc	cc	C	cc	C	C	cc	cc	Yes	Dex1
Open Lock	cc	cc	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Dex
Perform	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	Yes	Cha
Profession	cc	C	C	C	cc	C	C	C	C	C	C	No	Wis
Ride	C	cc	cc	C	C	cc	C	C	cc	cc	cc	Yes	Dex
Search	cc	cc	cc	cc	cc	cc	cc	C	C	cc	cc	Yes	Int
Sense Motive	cc	C	cc	cc	cc	C	C	cc	C	cc	cc	Yes	Wis
Sleight of Hand	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Dex1
Speak Language	cc	C	cc	cc	cc	cc	cc	cc	cc	cc	cc	No	None
Spellcraft	cc	C	C	C	cc	cc	cc	cc	cc	C	C	No	Int
Spot	cc	cc	cc	C	cc	C	cc	C	C	cc	cc	Yes	Wis
Survival	C	cc	cc	C	cc	cc	cc	C	cc	cc	cc	Yes	Wis
Swim	C	C	cc	C	C	C	cc	C	C	cc	cc	Yes	Str2
Tumble	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	No	Dex1
Use Magic Device	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Cha
Use Rope	cc	cc	cc	cc	cc	cc	cc	C	C	cc	cc	Yes	Dex

1 Armor check penalty applies to checks.

2 Double the normal armor check penalty applies to checks.