RPG Character Creation Project

John P. Baugh, Ph.D.
The Complete C++ Developer Course

What is this project about?

You are tasked with creating the foundational classes to represent the Player character, that is, the entity within the game world that the user of the game software would control. A **role-playing game** or **RPG** is a game in which you take on the persona (sometimes also called the avatar) of an in-game character, who goes on quests, fights enemies, looks for gold, crawls through dungeons or any other number of tasks, depending on the game.

Instructions

The Player Base Class

Player
- name : string - race : Race - hitPoints : int - magicPoints : int
+ Player(name : string, race : Race, hitPoints : int, magicPoints : int) + virtual ~Player() = default + getName() : string + getRace() : Race + whatRace() : string + getHitPoints() : int + getMagicPoints() : int + setName(name : string) : void + setRace(race : Race) : void + setHitPoints(hitPoints : int) : void + setMagicPoints(magicPoints : int) : void + *attack() : string = 0*

First, you'll create your **RPGProject** Visual Studio project. Then, you must create a base class, **Player**, which will contain the name, race, hitPoints, and magicPoints for the player. It should provide a constructor that takes four parameters corresponding to all four of its fields. It should also provide getters and setters for all four of the fields.

These fields are common to all Player objects. However, Player will be an **abstract class**, because it will contain an **attack()** method that returns a string, which is a pure virtual method (pure virtual function). Therefore, the derived classes must implement this attack method. Note that HP stands for hitPoints, and MP stands for magicPoints.

As another clarification – the destructor for Player should be virtual and set to default. This ensures proper cleanup when deleting Player objects through base class pointers — a critical practice when using polymorphism.

As a Modern C++ Tip: Be sure to mark the `attack()` function in each derived class with the `override` keyword. This ensures the method properly overrides the base class's virtual method and avoids silent bugs caused by mismatches.

The Derived Classes

Here are the specific player types that will be derived classes of the Player class:

- Warrior
  - 200 HP, 0 MP
  - Their attack method returns, “I will destroy you with my sword, foul demon!”
- Priest
  - 100 HP, 200 MP
  - Their attack method returns, “I will assault you with Holy Wrath!”
- Mage
  - 150 HP, 150 MP
  - Their attack method returns, “I will crush you with the power of my arcane missiles!”

Because most of their functionality is provided by the Player class, Player can have both specification and implementation files. But the implementations of the derived classes will be remarkably simple. So if you’d like, you don’t really need full separate implementation files for the Warrior, Priest, and Mage classes. You could put the empty body of the constructor for each derived class with a simple initialization list calling on the Player constructor – similar to what we did with runtime_error in the Exceptions section, inside the header file.

You can also put the implementation of the inherited virtual method, attack, inside the header file as well. But think carefully about what you need the user to pass in, and what you can take care of yourself. Remember, each of the derived classes have a set number for their HP and MP values.

An enum for the Race type

For this project, we're using **`enum class`** instead of traditional `enum` . It's the modern, preferred way to define enumerated types in C++. Unlike the older `enum`, **`enum class`** keeps values scoped (e.g., `Race::HUMAN` instead of just `HUMAN`), which prevents naming collisions and accidental conversions to int. It may feel slightly more verbose, but it's safer and a better long-term habit.

The race will be represented by an `enum class`, which will have the following values:

```
enum class Race { HUMAN, ELF, DWARF, ORC, TROLL };
```

The `enum` may be coded in the Player.h file, but outside of the Player class itself. This ensures all derived classes have access to it. You should provide an additional `whatRace()` method that returns a string representing the internal race of the `Player`. This is different from `getRace()`, which will return the actual enumerated value.

The main function

Inside main, you should allow the user to create different kinds of `Player` objects, allowing them to select the derived class, which you may **request as a profession from the user** from some simple menu. You should also let the user select the **race** of their character, and respond accordingly. Use a `vector<Player*>` to store your player objects. Each entry in the vector will point to a dynamically created object of type Warrior, Priest, or Mage. Since we're using polymorphism, the vector only needs to know the base type — Player*. Because of polymorphism, the player pointers can point to any of the derived class objects.

In modern C++ projects, we often use `unique_ptr<Player>` instead of raw pointers to manage memory automatically. We'll explore smart pointers in a future section, so just use regular raw pointers for this project.

The user interaction — menus, input, and output — will all be handled inside the main() function. Think of this part as a separate problem from building the class hierarchy. This modular approach will help keep your thinking clear and prevent you from feeling overwhelmed.

Once the user finishes creating their characters, iterate through the list of Player* pointers and display something like:

```
I'm a <Race> and my attack is: <Attack>
```