What you should do

- Use a cryptographically strong hashing function like bcrypt (see PHP's crypt() function).
- Use a random salt for each password.
- Use a slow hashing algorithm to make brute force attacks practically impossible.
- For bonus points, regenerate the hash every time a users logs in.

```
$username = 'Admin';
$password = 'gf45_gdf#4hg';

// A higher "cost" is more secure but consumes more processing power
$cost = 10;

// Create a random salt
$salt = strtr(base64_encode(mcrypt_create_iv(16, MCRYPT_DEV_URANDOM)),
'+', '.');

// Prefix information about the hash so PHP knows how to verify it later.
// "$2a$" Means we're using the Blowfish algorithm. The following two
digits are the cost parameter.
$salt = sprintf("$2a$%02d$", $cost) . $salt;

// Value:
// $2a$10$eImiTXuWVxfM37uY4JANjQ==

// Hash the password with the salt
$hash = crypt($password, $salt);

// Value:
// $2a$10$eImiTXuWVxfM37uY4JANjOL.oTxqp7WylW7FCzx2Lc7VLmdJIddZq
```

In the above example we turned a reasonably strong password into a hash that we can safely store in a database. The next time the user logs in we can validate the password as follows:

```
$username = 'Admin';
$password = 'gf45_gdf#4hg';

// For brevity, code to establish a database connection has been left out

$sth = $dbh->prepare('
  SELECT
    hash
  FROM users
  WHERE
    username = :username
  LIMIT 1
  ');

$sth->bindParam(':username', $username);

$sth->execute();

$user = $sth->fetch(PDO::FETCH_OBJ);

// Hashing the password with its hash as the salt returns the same hash
if ( hash_equals($user->hash, crypt($password, $user->hash)) ) {
  // Ok!
}
```

A few additional tips to prevent user accounts from being hacked:

- Limit the number of failed login attempts.
- Require strong passwords.
- Do not limit passwords to a certain length (remember, you're only storing a hash so length doesn't matter).
- Allow special characters in passwords, there is no reason not to.

That's it, happy coding!