Pointers

A pointer is a special type of variable that stores a memory address, not a value. It's used to reference existing variables so we can access them from somewhere else without duplicating them.

Open the project folder and run it in Visual Studio.

This project simulates a battle between the player and a random monster.

```
Player attacks the Vampire for 20 damage!
The Vampire has 150 health remaining...
Press any key to continue . . .
```

You can press any key to continue, and the monster/player will take turns attacking each other. The problem you'll notice is that their health never changes, resulting in an endless battle. This is due to a lack of pointers.

In the AttackMonster function, we send in the *defender* and *attacker* to calculate the damage taken. However, functions in C++ will send copies of the original through its parameters, not the original themselves. This means that when we reduce the health of the *defender*, we're only reducing the health of the copy; the original remains undamaged.

```
void AttackMonster(Monster defender, Monster attacker)
{
    defender.health -= attacker.damage;

    std::cout << attacker.name << " attacks the " << defender.name << " for "
        << attacker.damage << " damage!" << std::endl;
}
```

This is where pointers come in.

Pointers allow us to send a reference to the original variable so we can change its health properly. It's important to note that only the *defender* needs to be pointed to because we're changing its health. The *attacker* doesn't have any of its variables changed, so it's fine to leave as it is (though making it a pointer would save some memory).

To start, we need to make *current* a pointer so it correctly references whatever enemy it's assigned.

```
//pick a random enemy
Monster * current;
srand(time(0)); //this generate a random seed
int r = rand() % 3; //this generates a random number between 0-3 exclusive
switch (r)
{
    case 0: current = &dragon; break;
    case 1: current = &vampire; break;
    case 2: current = &zombie; break;
}
```

Add an asterisk (*) to the line where we declare *current*. This makes it a pointer: it longer contains a Monster variable, but instead is pointing towards a memory address containing a Monster.

Then add an ampersand (&) to the start of each Monster variable when we're assigning it. The ampersand grabs the memory address of a regular variable. Now, *current* is pointing towards the chosen monster.

For sake of clarity, from now on let's assume the monster chosen is *dragon*.

Now the *AttackMonster* and *DisplayMonsterHealth* functions are going to complain because they've so far been looking for regular variables, not pointers. So let's change that.

In both the declaration and definition, change *AttackMonster* to the following:

```
void AttackMonster(Monster * defender, Monster attacker);
```

This means that defender needs to be a pointer.

Inside the function definition, we need to change the code slightly. We can't access member variables from a pointer with a period (.) like we usually do, we need to use the arrow (->) instead.

```
 defender->health -= attacker.damage;
```

The arrow points towards the original and accesses its members. So we're not changing *defender* or *current*, we're changing *dragon.*

```
//fight until someone is defeated
while (current.health > 0 && player.health > 0)
{
    //player attacks current monster
    AttackMonster(current, player);
    DisplayMonsterHealth(current);

    //wait for input then clear screen
    system("pause");
    system("cls");

    //current monster attacks player
    AttackMonster(player, current);
    DisplayMonsterHealth(player);
```

The last few steps are to fix the remaining errors inside the while loop.

I'll leave you to figure this part out yourself but remember the following:

- *current* is a pointer, *player* is a regular variable
- To get the variable from a pointer we use an asterisk
- To get a memory address from a variable we use an ampersand
- To access a member from a pointer, we use an arrow instead of a period

Once it's working you should be able to run the program and watch the player and monster gradually reduce each other's health.

As a bonus challenge, try to implement multiple players (like a Fighter, Barbarian, or Rogue) with different health and attack values.

In the same way we generate a random monster, generate a random player, then use that for the battle simulation instead of the regular player.