Tutorial – STL Algorithms: Zorp part 12

Introduction:

In the previous tutorial, we added an inventory system to the *Player* class so that the player could pick up and store powerups.

In this tutorial, we will use the STL sort algorithm to sort the collected items alphabetically by name.

Requirements:

You will need to have completed the following previous tutorials:

- Zorp Part 1: Variables. Available under the topic *Variables and Constants.*
- Zorp Part 2: Constants. Available under the topic *Variables and Constants.*
- Zorp Part 3: Arrays. Available under the topic *Arrays.*
- Zorp Part 4: Conditional Statements. Available under the topic *Conditional Statements*.
- Zorp Part 5: Loops. Available under the topic *Loops*.
- Zorp Part 6: Functions. Available under the topic *Functions*.
- Zorp Part 7: Character Arrays. Available under the topic *Arrays*.
- Zorp Part 8: Structures. Available under the topic *Structures and Classes*.
- Zorp Part 9: Classes. Available under the topic *Structures and Classes*.
- Zorp Part 10: Constructors and Destructors. Available under the topic *Structures and Classes*.
- Zorp Part 11: STL Vector. Available under the topic *Standard Template Library*.

Sorting the Powerups:

Our collection of powerups in the *Player* class is implemented as a *std::vector*. This means that we can use the STL *sort()* algorithm to sort the items.

The logical time and place to sort the powerups the player has collected is when the player picks up a new powerup.

This happens in the *Player* class's *pickup()* function (which is called in response to the *executeCommand* processing a 'pick up' command from the player).

Open the *Player.cpp* file and modify this function as follows:

```
bool Player::pickup(int roomType) {
	static const char itemNames[15][30] = {
		"indifference", "invisibility", "invulnerability", "incontinence",
		"improbability", "impatience", "indecision", "inspiration",
		"independence", "incurability", "integration", "invocation",
		"inferno", "indigestion", "inoculation"
	};

	int item = rand() % 15;
	char name[30] = "";

	switch (roomType) {
		case TREASURE_HP:
			strcpy(name, "potion of ");
			break;
		case TREASURE_AT:
			strcpy(name, "sword of ");
			break;
		case TREASURE_DF:
			strcpy(name, "shield of ");
			break;
		default:
			return false;
	}

		// append the item name to the string
	strncat(name, itemNames[item], 30);
	std::cout << EXTRA_OUTPUT_POS << RESET_COLOR << "You pick up the "
		<< name << std::endl;
	m_powerups.push_back(Powerup(name, 1, 1, 1.1f));

	std::sort (m_powerups.begin(), m_powerups.end(), Powerup::compare);

	std::cout << INDENT << "Press 'Enter' to continue.";
	std::cin.clear();
	std::cin.ignore(std::cin.rdbuf()->in_avail());
	std::cin.get();
	return true;
}
```

We've inserted just one line of code that will all the STD *sort()* function and sort our *vector* of *Powerup* objects for us.

We pass in the begin and end iterators of our *m_powerups* vector, and a *function pointer* to the custom comparison function we want to use to compare instances of the *Powerup* class. (Don't worry, we cover function pointers in much more detail later in the course. For now, it's only important to understand how to use them, not why they work).

If you try to compile this you should notice we get an error. The *Powerup::compare* function doesn't exist yet.

But why do we even need a custom comparison function anyway?

If we were storing integers, floats, or some other basic data type in our vector, then C++ already knows how to compare them. It just uses the *less than operator*.

But C++ doesn't know how to compare two instances of the *Powerup* class. What makes one power up "less than" another one? Should they be compared by their name? Or do we want to compare them based on how much they modify the player's HP, AT, or DF parameters? And even if C++ knew we wanted to sort them by name, it still doesn't know *how* to do that (we need to use the *strcmp()* function, and not the *less than operator*).

All this means that the only way we can tell the *sort()* function *how* to compare the *Powerup* class objects is to give it a custom function it can use to perform this comparison. Any time *sort()* compares two power ups, it will use this function to do it.

The Comparison Function:

While we *could* write a global function to compare two *Powerup* objects, it makes much more sense to make this comparison function a member function of the *Powerup* class.

We need to make this function *static*, because of the way the *sort()* function will use it. It will pass in the two *Powerup* objects it wants compared as input arguments, as opposed to calling the function on one object and passing in the other.

That was probable a little hard to understand, so here's the difference in code:

```
Powerup p1, p2;

// we need to do this:
Powerup::compare(p1, p2);

// not this:
p1.compare(p2);
```

The difference is subtle, but important.

In the *Powerup* header, define the new comparison function as follows:

```
class Powerup {
public:
	Powerup(const char name[30], float health, float attack, float defence);
	~Powerup();

	char* getName();
	float getHealthMultiplier();
	float getAttackMultiplier();
	float getDefenceMultiplier();

	static bool compare(const Powerup& p1, const Powerup& p2);

private:
	char m_name[30];
	float m_healthMultiplier;
	float m_attackMultiplier;
	float m_defenceMultiplier;
};
```

Because we want our powerups to be sorted alphabetically, we need to write a comparison function that will return *true* if the name of the first powerup is less than that of the second.

Remember that when we're comparing strings, we can't use the less than sign. In C++ this would actually compare the memory addresses of where the two strings are stored (although it is a valid way to compare two strings in some other programming

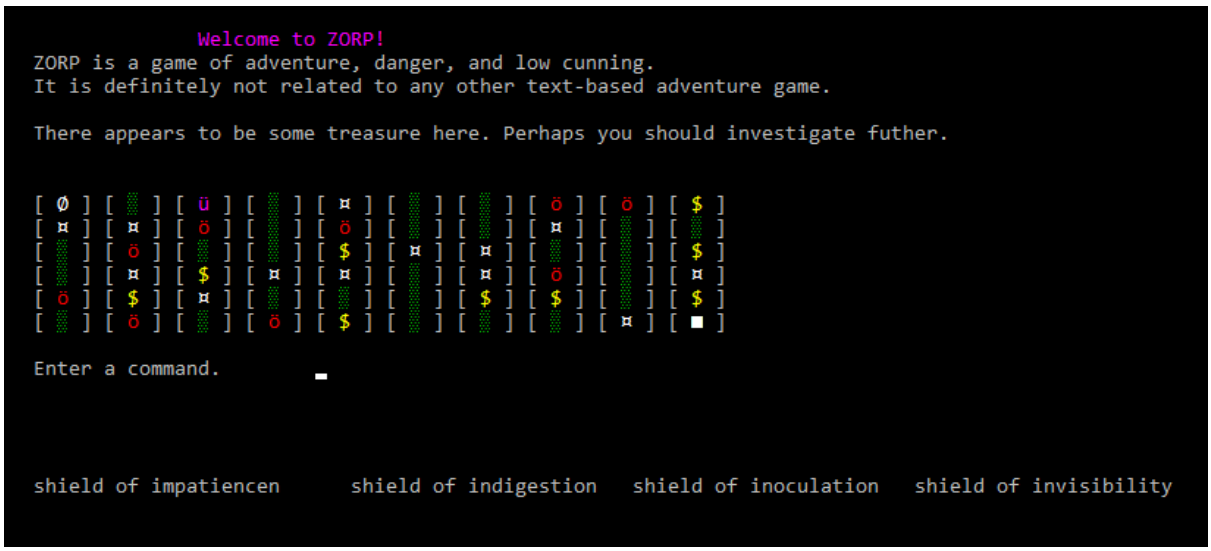
The implementation of the comparison function is as follows:

```
bool Powerup::compare(const Powerup & p1, const Powerup & p2)
{
	return (strcmp(p1.m_name, p2.m_name) < 0) ? true : false;
}
```

The important thing to note with this function is we need to return *false* if the second value is the same as, or greater than the first.

If your comparison function is incorrect, then in the best case your results will not be sorted correctly. In the worst case you will actually cause an error inside the *sort()* function and your program will crash.

Compile and execute your game now. You should see that the powerups the player collects are written to the console in alphabetical order:



While the example in this tutorial was relatively contrived, you could extend this concept and use the STL sort function to do things like maintain an ordered list of high scores, or to sort a list of enemies according to their distance from the player.