Tutorial – Constructors and Destructors: Zorp part 10

Introduction:

In this tutorial we will update our class constructors to initialize the game objects upon creation.

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*.

Constructors:

In our previous tutorial, we defined a default constructor and an overloaded constructor for the *Player* class:

```
Player::Player()
{
	m_mapPosition.x = 0;
	m_mapPosition.y = 0;
}

Player::Player(int x, int y)
{
	m_mapPosition.x = x;
	m_mapPosition.y = y;
}
```

You will notice that the only thing happening inside the body of these constructor functions is the initialization of class member variables.

Because this type of constructor is so common, C++ gives us a quick way to initialize member variables through the use of member initialization lists.

Let's write the same code using member initialization lists:

```
Player::Player() : m_mapPosition(Point2D{ 0, 0 })
{
}

Player::Player(int x, int y) : m_mapPosition(Point2D{ x, y })
{
}
```

You can now see that the body of the function is completely empty, and our member variables have been initialized after the constructor argument list.

If we have more than one member variable to initialize, we can separate them using commas:

```
Room::Room() : m_type(EMPTY), m_mapPosition(Point2D{0, 0})
{
}
```

The benefit of using this type of initialization is that the compiler is able to optimize the compiled code. Essentially, the program does not need to copy the value onto the stack and call the member's assignment operator when initializing the member variable.

This is a very common way that C++ programmers write their constructors, so it is useful to become familiar with this syntax.

Uniform Initialization in C++ 11:

When we use rounded brackets in the member initialization list, this is called *direct initialization.*

```
Room::Room() : m_type(EMPTY) // directly initialize member variable
{
}
```

In C++11, *uniform initialization* can be used:

```
Room::Room() : m_type{EMPTY} // uniformly initialize member variable
{
}
```

When your compiler is C++11 compliant, you should favour uniform initialization over direct initialization.

Let's take a look at one benefit of uniform initialization.

In our *Player* class constructor, we are initializing a *Point2D* structure:

```
Player::Player() : m_mapPosition(Point2D{ 0, 0 })
{
}

Player::Player(int x, int y) : m_mapPosition(Point2D{ x, y })
{
}
```

But why do we need to type *Point2D*? The compiler already knows that the *m_mapPosition* variable is a *Point2D* structure, so shouldn't we just be able to pass it the values without explicitly saying we want to create an instance of the *Point2D* structure?

We can update our *Player* constructors using uniform initialization to achieve this:

```
Player::Player() : m_mapPosition{ 0, 0 }
{
}

Player::Player(int x, int y) : m_mapPosition{ x, y }
{
}
```

We can update our other class constructors too:

```
Room::Room() : m_type{ EMPTY }, m_mapPosition{ 0, 0 }
{
}

Game::Game() : m_gameOver{ false }
{
}
```