



# **Starter's Development Guide**

Vol. I

Sarah Alexander

GitHub - <https://github.com/Nyeriah>

Starter's Development Guide  
Vol. I

## Summary

1. Introduction to Volume I.
2. TrinityCore, The Power of the Community
3. Knowledge Requirements to Develop TrinityCore
4. Scripting
  - 4.1 Scripting of different kinds of objects
  - 4.2 Event Hooks
  - 4.3 Functions
  - 4.4 Code Structure and Conventions
5. Practice: Scripting a simple creature
6. Considerations and Conclusions about Volume I.

## 1. Introduction to Volume I

In this first volume, we will be going through the basics of what the TrinityCore project is and how we can get started with it, acknowledging the required skills necessary prior developing and learning some of the basics of scripting.

If you're a development enthusiast and would like to contribute to the TrinityCore project but you don't quite know where to start yet, you may want to check out this guide.

The text here presented does not aim to and does not solve every of your doubts. We'll going through this step by step. If you're already an advanced developer or you have already done something on your own, this volume will offer you nothing new – skip it.

Thorough this guide we will be using the 3.3.5 branch of TrinityCore as the base study subject.

## 2. TrinityCore, The Power of the Community

TrinityCore is a vast and complex learning project that provides a MMORPG framework, built using C++ and using the back-end of MySQL as described in the official TrinityCore website.

**Community contribution is the life of any OpenSource project**, and that's probably one of TrinityCore's strengths.

Over time, the dedicated team of developers and the community have worked together to solve thousands of issues, constantly improving the framework's performance and utility. Together, sharing their knowledge and work, the community has worked to build something consistent and qualitative.

Interested in helping TrinityCore? There are several ways to help. If you are not a developer, you can always try to set up your local server anyway and help reporting, confirming or denying issues. However, if you have already done a thing or two on your own and they are valid changes, you can always propose a pull request at the main repository. We will always be more than glad to receive your contribution and assist you through any issues you may find.

Visit TrinityCore's official website (**[link to official website](https://www.trinitycore.org/)**: <https://www.trinitycore.org/>) if you would like check out more information about it.

### 3. Knowledge Requirements to Develop TrinityCore

First things first, before you start working with TrinityCore you should acknowledge your skills and know how far you can push your limits. Have you worked with any programming language before? Do you have sense of programming logic? Have you managed a database before? If you did all of these things, setting your own server and starting to get things done will be a piece of cake.

If you never had any contact with any programming language or never managed a database, it's advisable to read some sources on the subject of C++ programming and MySQL database management before proceeding, otherwise the understanding of this guide and even the setting of your local server itself may prove to be difficult.

It's not really required to master the C++ programming language to be productive, neither is it required that you know all of the SQL commands by the top of your mind. It's required however that you have some basic knowledge of **programming logic and C++ syntax** if you're planning to meddle with core or any of the scripts. It's also not required for you to master everything about SQL but you should know the basic management query commands (**INSERT, DELETE, UPDATE**).

Even if you're not really fond of C++, there's a whole lot of things that can be done just through the usage of the database and its scripting engine (**SmartScripts**). We'll be covering it at some point down the road.

If you consider yourself ready and wishes to proceed, you may try out setting your own server now if you haven't already. This guide will not cover the installation process because it is already well-detailed at the TrinityCore wiki (**link to the installation article**: <https://trinitycore.atlassian.net/wiki/display/tc/Installation+Guide>).

## 4. Scripting

### 4.1 Scripting of Different Kinds of Objects

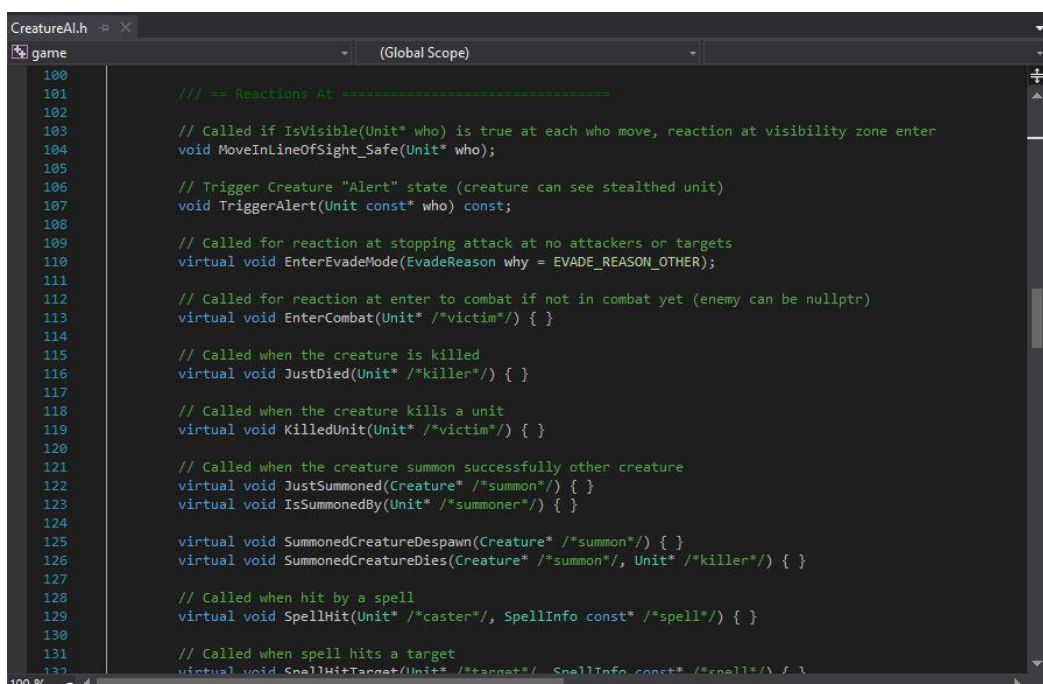
There are several game entities that can be scripted, for example, creatures, game objects, items, spells, instances, battlegrounds and even players!

Each of them has different kinds of **event hooks, functions and properties** depending on the class they inherit.

Scripts are instances of objects, in other words, the current running script is only valid to the entity it's associated to. So, if I spawn two instances of the same scripted creature, each will execute their scripts independently of the other.

### 4.2 Event Hooks

Event hooks are pieces of code executed whenever an event happens. For example, when a scripted **creature receives damage**, the **DamageTaken()** hook is executed. To check for a list of all the possible event hooks, check the virtual voids at the class your script is inheriting (i.e **CreatureAI**). **Note: when used in scripts, event hooks must be followed by the “override” keyword.**



```
100
101 // == Reactions At ==
102
103 // Called if IsVisible(Unit* who) is true at each who move, reaction at visibility zone enter
104 void MoveInLineOfSight_Safe(Unit* who);
105
106 // Trigger Creature "Alert" state (creature can see stealthed unit)
107 void TriggerAlert(Unit const* who) const;
108
109 // Called for reaction at stopping attack at no attackers or targets
110 virtual void EnterEvadeMode(EvadeReason why = EVADE_REASON_OTHER);
111
112 // Called for reaction at enter to combat if not in combat yet (enemy can be nullptr)
113 virtual void EnterCombat(Unit* /*victim*/) { }
114
115 // Called when the creature is killed
116 virtual void JustDied(Unit* /*killer*/) { }
117
118 // Called when the creature kills a unit
119 virtual void KilledUnit(Unit* /*victim*/) { }
120
121 // Called when the creature summon successfully other creature
122 virtual void JustSummoned(Creature* /*summon*/) { }
123 virtual void IsSummonedBy(Unit* /*summoner*/) { }
124
125 virtual void SummonedCreatureDespawn(Creature* /*summon*/) { }
126 virtual void SummonedCreatureDies(Creature* /*summon*/, Unit* /*killer*/) { }
127
128 // Called when hit by a spell
129 virtual void SpellHit(Unit* /*caster*/, SpellInfo const* /*spell*/) { }
130
131 // Called when spell hits a target
132 virtual void SpellHitTarget(Unit* /*target*/, SpellInfo const* /*spell*/) { }
```

Event hooks often pass objects through their parameters, we can use them to manipulate these objects in our script. Check out the example.

```
// Example 1
void EnterCombat(Unit* who) override
{
    // In this example, we use the object passed by the event hook's parameter
    who->KillSelf(); // The first unit to enter in combat with this creature will suicide... grim.
}

// Example 2
void EnterCombat(Unit* /*who*/) override
{
    // In this example, we do not use the object passed by the event hook's parameter
    // So we comment it out to prevent "Unused variable" compile warnings.

    me->KillSelf(); // "me" = pointer to object we are manipulating
                  // the "creature" we are scripting,
                  // If you're a developer, it would be something alike "this->" calls.
}

// NOTE: YOU CANNOT OVERRIDE AN EVENT HOOK TWICE, it'll return you a compile error.
```

**Remember to comment out the variable passed through the parameter if you are not using it, that will suppress compiler warnings.**

Also note that you may not override an event hook twice in a script, that will throw you errors.

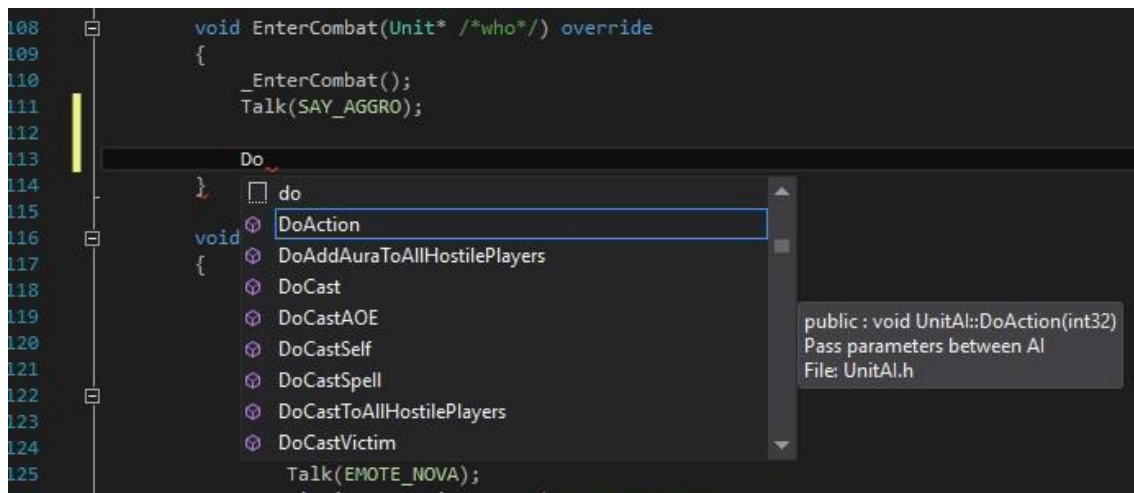
Some event hooks execute default code when not overridden, so overriding them may be useful if the code they execute causes unintended behavior. However, if you still wish their native code to be executed, you can just call the event hook function from the parent class, like in the example below.

```
void EnterCombat(Unit* who) override
{
    ScriptedAI::EnterCombat(who);
}
```



### 4.3 Functions

In the same fashion as the Event Hooks, several functions are inherited from the parent class used in the script. Functions represent blocks of code that are executed whenever the function is called. If you're using the Visual Studio IDE, the intellisense should be extremely helpful to track the array of functions at our disposal. Visual Studio: **Pressing F2 while having a function selected will also navigate it to its source**, you can check what each of them do that way.



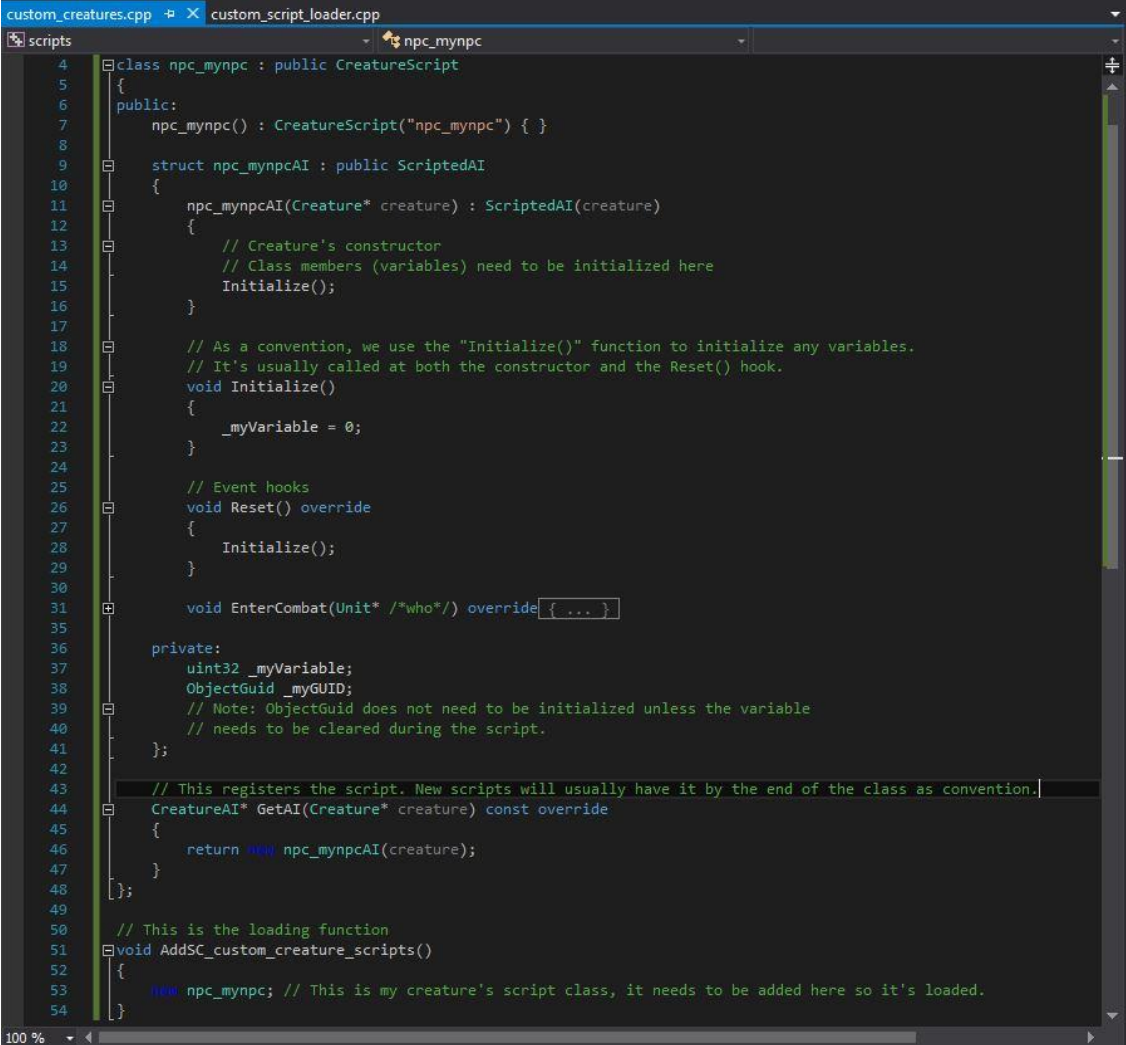
Functions may have parameters that must be filled when used and may also have several overloads (the same function, interpreted in a different way, processing a different block of code). For example, DoCast() is the most generic function when casting spells. It accepts a **target** (*Unit\** object, which means it can either be a player or creature) and a **spellId** (uint32 object) parameter. Simply put, target is the target that will be used when the spell is cast, and spellId is the spell that will be cast. Check each function individually to ensure it's really the one you need to get the job done, it helps keeping the code clean and consistent.

## Properties

Object properties are in general information gotten from the templates (creature template, object template, so on) and other specific information about the given object (for example, current health, current mana, current position). They are usually accessed by the "GetVariable()" functions. If you are familiar to variable encapsulation you should get the idea. But if you are not, these are functions that return these values, for example, ***GetMaxHealth()*** will return the creature's total health, as an uint32 object type.

## 4.4 Code Structure and Conventions

The image below pictures the model of a basic creature script. The structure may change depending on the entity you are scripting, but in general, it does not stray away far from this. The image also presents some of the TrinityCore conventions.



```
custom_creatures.cpp  custom_script_loader.cpp
scripts
  npc_mynpc
4  class npc_mynpc : public CreatureScript
5  {
6  public:
7      npc_mynpc() : CreatureScript("npc_mynpc") { }
8
9      struct npc_mynpcAI : public ScriptedAI
10     {
11         npc_mynpcAI(Creature* creature) : ScriptedAI(creature)
12         {
13             // Creature's constructor
14             // Class members (variables) need to be initialized here
15             Initialize();
16         }
17
18         // As a convention, we use the "Initialize()" function to initialize any variables.
19         // It's usually called at both the constructor and the Reset() hook.
20         void Initialize()
21         {
22             _myVariable = 0;
23         }
24
25         // Event hooks
26         void Reset() override
27         {
28             Initialize();
29         }
30
31         void EnterCombat(Unit* /*who*/) override { ... }
32
33     private:
34         uint32 _myVariable;
35         ObjectGuid _myGUID;
36         // Note: ObjectGuid does not need to be initialized unless the variable
37         // needs to be cleared during the script.
38     };
39
40     // This registers the script. New scripts will usually have it by the end of the class as convention.
41     CreatureAI* GetAI(Creature* creature) const override
42     {
43         return new npc_mynpcAI(creature);
44     }
45 };
46
47 // This is the loading function
48 void AddSC_custom_creature_scripts()
49 {
50     new npc_mynpc; // This is my creature's script class, it needs to be added here so it's loaded.
51 }
```

## Variable Conventions

You can name your variables in any way you wish but it's a good practice to give some relevant name to them. Also, it's common to specify it in the name if they are a GUID variable, for example, "playerGUID". TrinityCore also follows the following variable naming convention for what concerns the variable naming, depending on the variable scope:

- Public: MyVariable;
- Protected: MyVariable;
- Private: \_myVariable;
- Local: myVariable

## Pointers!

**Never store pointers to world objects (creatures, gameobjects, players, items) on class members**, store their GUID instead. These objects can become invalid pointers and that will result in a crash when they're accessed. To access them, use the "ObjectAccessor" namespace.

## Initialize your class members!

TrinityCore uses the "Initialize()" method to initialize class members as a convention. If you're using more than a single variable, it's recommended to use it when initializing your variables in the constructor and "Reset()" hook (depending if it fits what you want to do in your script or not).

## 5. Practice: Scripting a Simple Creature

Through this volume we learned a bit about the scripting, event hooks and functions. Now it's about time we go and try to script something ourselves: a quite simple creature.

**Objective:** Further understand how event hooks work, use some of the functions available.

**Script:** For learning purposes, we will create a custom creature and assign to it a core script. In this script, the following behavior is expected:

- The creature will say a line of text when the player moves in its line of sight.
- The creature will follow the player around when it receives the /followme text emote.
- The creature will stand still when it receives the /wait text emote.
- The creature will return to its home position when it receives the /bye text emote.

For convenience, this is the SQL for the custom creature I have created.

[illegible]

To create your own custom creature, just insert a new row at the `creature\_template` table. Keep in mind you'll have to follow the rules that may apply to certain columns to have the results you expect. For more information, check the `creature\_template` wiki page at [https://trinitycore.atlassian.net/wiki/display/tc/creature\\_template](https://trinitycore.atlassian.net/wiki/display/tc/creature_template)

To script our creature, we'll be using the basic structure we used in the previous example, only renaming the class to fit our new creature.

```
class npc_high_priestess_lana : public CreatureScript
{
public:
    npc_high_priestess_lana() : CreatureScript("npc_high_priestess_lana") { }

    struct npc_high_priestess_lanaAI : public ScriptedAI
    {
        npc_high_priestess_lanaAI(Creature* creature) : ScriptedAI(creature)
        {
        }
    };

    CreatureAI* GetAI(Creature* creature) const override
    {
        return new npc_high_priestess_lanaAI(creature);
    }
};

// This is the loading function
void AddSC_custom_creature_scripts()
{
    new npc_mynpc; // This is my creature's script class, it needs to be added here so it's loaded.
    new npc_high_priestess_lana;
}
```

Later we'll be adding the script hooks that we will be using during our practice.

- **Void Reset():** Called when the creature reaches home or respawns. We use it to reset variables (we'll be using one variable).
- **Void MoveInLineOfSight(Unit\* who):** Called whenever an unit moves in line of sight of the creature, sight range is quite large and it can be triggered more than once as the unit moves towards it. Keep in mind that units can be either players or creatures.
- **Void ReceiveEmote(Player\* player, uint32 textEmote):** Called whenever the creature receives a chat emote (/wave, /followme, /bye among others).

Let's start off by adding the the MoveInLineOfSight() hook:

```
void MoveInLineOfSight(Unit* who) override
{
    if (!_hasSeenPlayer) // Here we prevent it from firing more than once by using a bool variable.
    {
        // GetTypeId() returns the world object type.
        // Since we know Unit can be either a player or creature, we check it to
        // prevent it from firing at some random creature, pet or whatever.
        // Also check for the distance, so we don't fire it when we are too far away.
        if (who->GetTypeId() == TYPEID_PLAYER && me->IsWithinDistInMap(who, 15.0f))
        {
            // This is actually bad practice, but it works wonders while debugging
            // and it's quite easy to use.
            // Since we won't be converging this in depth at once, it should be fine.
            // Normally, texts said by creatures should use the creature_text table at database instead.
            me->Say("Champion! Over here!", LANG_UNIVERSAL);

            // Let's do some fun stuff... wave to player and face it.
            me->SetFacingToObject(who);
            me->HandleEmoteCommand(EMOTE_ONESHOT_WAVE);

            _hasSeenPlayer = true;
        }
    }
}

private:
    bool _hasSeenPlayer;
};
```

There are some peculiarities with this hook that we should keep in mind and that I'll be addressing through the code.

**It can be triggered by either creatures or players.** In our case, we only want it to be triggered by players.

**Creatures have a long sight range.** We don't wish the creature to trigger the greeting event too soon, otherwise the player won't be able to see it, so we are checking if the player is within 15 yards.

**It can be triggered multiple times as the unit moves.** We only wish the creature to greet us once, so we are using a bool variable to prevent it from happening again.



The **Say()** function is generally not recommended for any serious scripting, you'll want to properly use **Talk()** and the ``creature_text`` table content instead. But in our case, since this is written for learning purposes, it'll do just fine. It's also great for debugging simple things.

**SetFacingToObject()** forces the creature to face the object, sets its orientation so it faces it visually.

**HandleEmoteCommand()** makes the creature play an emote, like waving, or applauding. The emotes are defined at the emote enum generally and the elements are usually named **EMOTE\_ONESHOT\_\*\*\*\***. "ONESHOT" stands for the fact they are not emotes that repeat.

Next, we'll be moving to the `ReceiveEmote()` hook, and that's where most of our code will be at.

```
// This hook is called whenever the creature receives a chat emote from a player.
void ReceiveEmote(Player* player, uint32 textEmote) override
{
    if (!_hasSeenPlayer) // Prevent starting anything before we are introduced
        return;

    switch (textEmote)
    {
        case TEXT_EMOTE_FOLLOW:
            me->SetFacingToObject(player);
            // This function plays emotes
            me->HandleEmoteCommand(EMOTE_ONESHOT_TALK);
            me->Say("I'll be following you, champion. We must act quickly, "
                "the agents of the Burning Legion must not set a foothold at Karazhan.", LANG_UNIVERSAL);
            // GetMotionMaster() returns the movement generator, and eventually all of the
            // movement related functions.
            // In our case, we want our NPC to follow us around when we tell them to.
            me->GetMotionMaster()->MoveFollow(player, 5.0f, 0.0f);

            DoCastSelf(SPELL_FORTITUDE);
            DoCast(player, SPELL_FORTITUDE);
            break;

        case TEXT_EMOTE_WAIT:
            me->SetFacingToObject(player);
            me->HandleEmoteCommand(EMOTE_ONESHOT_QUESTION);
            me->Say("Are you sure you wish to proceed alone? "
                "Very well then, I will wait here.", LANG_UNIVERSAL);
            // This will clear our previous MoveFollow() call.
            me->GetMotionMaster()->Clear();
            break;

        case TEXT_EMOTE_BYE:
            me->SetFacingToObject(player);
            me->Say("I'll head back to the entrance. I'll be there if you require my assistance.", LANG_UNIVERSAL);
            me->GetMotionMaster()->Clear();
            // Enter evade mode - leave combat, move "home" (or spawn position), remove all auras
            EnterEvadeMode();
            break;

        default:
            break;
    }
}
```

Here we are basically just using the same functions we used in the previous hook, however we're also issuing motion calls.

**GetMotionMaster()** returns the creature's movement generators. From there, you can issue several calls, like **MoveChase()**, **MoveFollow()**, **MoveIdle()** and so on. **Clear()** clears any previously assigned movement call.

**DoCast(target, spell)** casts a spell on the player issuing the emote, just a cosmetic thing. **DoCastSelf()** casts the spell on the creature itself. The spell entry is defined in an enum as per TrinityCore standards – try to avoid “magic numbers” at all costs.

In the next pages I'll be posting the entire source code, if you wish to try it out you can just copy it in your solution's file. Through this guide I'll be using the custom\_creatures.cpp file I created.

Source: npc\_high\_priestess\_lana  
Creature script, ScriptedAI  
Development Guide Volume: 1, Practice: 1

**Trivia:** *As the Burning Legion threatens Azeroth once again, a group of champions was sent to investigate the forgotten catacombs of Karazhan seeking the sources of demonic energy. The Blood Elves of Silvermoon have sent their most formidable priestess, Lana, The High Priestess of Silvermoon, to assist the champions in their journey. Who knows what sorts of evil magic and demonic beasts will they find lurking in the shadows?*

## Starter's Development Guide Vol. I

```
enum Lana
{
    SPELL_FORTITUDE = 48161
};

class npc_high_priestess_lana : public CreatureScript
{
public:
    npc_high_priestess_lana() : CreatureScript("npc_high_priestess_lana") { }

    struct npc_high_priestess_lanaAI : public ScriptedAI
    {
        npc_high_priestess_lanaAI(Creature* creature) :
ScriptedAI(creature)
        {
            Initialize();
        }

        void Initialize()
        {
            _hasSeenPlayer = false;
        }

        void Reset() override
        {
            Initialize();
        }

        // As the name suggests, this hook is called whenever an unit
(player or creature)
        // moves in the creature's line of sight.
        void MoveInLineOfSight(Unit* who) override
        {
            if (!_hasSeenPlayer) // Here we prevent it from firing more
than once by using a bool variable.
            {
                // GetTypeId() returns the world object type.
                // Since we know Unit can be either a player or
creature, we check it to
                // prevent it from firing at some random creature, pet
or whatever.
                // Also check for the distance, so we don't fire it
when we are too far away.
                if (who->GetTypeId() == TYPEID_PLAYER && me-
>IsWithinDistInMap(who, 15.0f))
                {
                    // This is actually bad practice, but it works
wonders while debugging
                    // and it's quite easy to use.
                    // Since we won't be converging this in depth at
once, it should be fine.
                    // Normally, texts said by creatures should use
the creature_text table at database instead.
                    me->Say("Champion! Over here!",
LANG_UNIVERSAL);

                    // Let's do some fun stuff... wave to player
and face it.
                    me->SetFacingToObject(who);
```

## Starter's Development Guide

### Vol. I

```
        _hasSeenPlayer = true;
    }
}

// This hook is called whenever the creature receives a chat emote
from a player.
void ReceiveEmote(Player* player, uint32 textEmote) override
{
    if (!_hasSeenPlayer) // Prevent starting anything before we
are introduced
        return;

    switch (textEmote)
    {
        case TEXT_EMOTE_FOLLOW:
            me->SetFacingToObject(player);
            // This function plays emotes
            me->HandleEmoteCommand(EMOTE_ONESHOT_TALK);
            me->Say("I'll be following you, champion. We
must act quickly, "
                    "the agents of the Burning Legion must
not set a foothold at Karazhan.", LANG_UNIVERSAL);
            // GetMotionMaster() returns the movement
generator, and eventually all of the
            // movement related functions.
            // In our case, we want our NPC to follow us
around when we tell them to.
            me->GetMotionMaster()->MoveFollow(player, 5.0f,
0.0f);

            DoCastSelf(SPELL_FORTITUDE);
            DoCast(player, SPELL_FORTITUDE);
            break;
        case TEXT_EMOTE_WAIT:
            me->SetFacingToObject(player);
            me->HandleEmoteCommand(EMOTE_ONESHOT_QUESTION);
            me->Say("Are you sure you wish to proceed
alone? "
                    "Very well then, I will wait here.",
LANG_UNIVERSAL);
            // This will clear our previous MoveFollow()
call.
            me->GetMotionMaster()->Clear();
            break;
        case TEXT_EMOTE_BYE:
            me->SetFacingToObject(player);
            me->Say("I'll head back to the entrance. I'll
be there if you require my assistance.", LANG_UNIVERSAL);
            me->GetMotionMaster()->Clear();
            // Enter evade mode - leave combat, move "home"
(or spawn position), remove all auras
            EnterEvadeMode();
            break;
        default:
            break;
    }
}
```

## Starter's Development Guide Vol. I

```
private:
    bool _hasSeenPlayer;
};

CreatureAI* GetAI(Creature* creature) const override
{
    return new npc_high_priestess_lanaAI(creature);
};

// This is the loading function
void AddSC_custom_creature_scripts()
{
    new npc_high_priestess_lana;
}
```

## Starter's Development Guide Vol. I

Here are some images of the results we get with this code in-game.





## Starter's Development Guide Vol. I



<https://gaming.youtube.com/watch?v=fixdUiptFpU&feature=share>

**Remember:** it's important to assign the script to the creature at the database. The `creature\_template` ScriptName column must match the name you gave to your script at core (the name within quotes at the class constructor).

## 6. Considerations and Conclusions about Volume I

This volume covered most of what is required to start developing things using TrinityCore, following the TrinityCore coding standards.

The items below are the goals I have tried to make the reader achieve through this volume:

- Know most of the TrinityCore coding conventions.
- Know what event hooks are and how do they work.
- Know what functions are and how the most common ones work.
- Know how scripts are structured and how to write one following the TrinityCore standards
- Know how to register a new script and link it to a creature.

After reading through this, you should be ready to go on and script your own creature. This volume does not cover creature scripting in depth but it serves as the starting point that we'll be using through this project – from here we'll be covering more complex scripts.

In the next volume, I'll be going over creature AI more closely, working with timed events and combat mechanics.



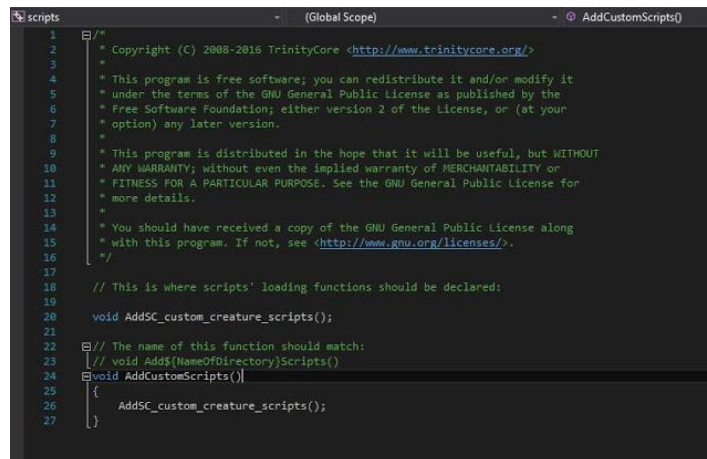
## Extra: Adding a custom file to your solution

Adding a new file to a Visual Studio solution is a rather simple task - you can either right click and add a new C++ item, or copy-paste an existing file and then drag it to your solution.

However, there are a few things you must do in order to have your scripts load properly. By default, TrinityCore already has a "Custom" folder reserved to be used by custom content. You can use other folders if you wish, but there's no reason for us to trouble ourselves with that.

1. Create your script file in the scripts/custom folder and add it to your solution.
2. Create a loading function in it ("AddSC\_filename()") where "filename" is the name of your file.
3. Add the references of the loading functions you just created to the scripts/custom/custom\_script\_loader.cpp file.

I have added a custom script file named "custom\_creature\_scripts" to my custom scripts folder, so this is how my loader file looks like:



```
1  //
2  * Copyright (C) 2008-2016 TrinityCore <http://www.trinitycore.org/>
3  *
4  * This program is free software; you can redistribute it and/or modify it
5  * under the terms of the GNU General Public License as published by the
6  * Free Software Foundation; either version 2 of the License, or (at your
7  * option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful, but WITHOUT
10 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
11 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
12 * more details.
13 *
14 * You should have received a copy of the GNU General Public License along
15 * with this program. If not, see <http://www.gnu.org/licenses/>.
16 */
17
18 // This is where scripts' loading functions should be declared:
19 void AddSC_custom_creature_scripts();
20
21 // The name of this function should match:
22 // void AddSC(NameOfDirectory)Scripts()
23 void AddCustomScripts()
24 {
25     AddSC_custom_creature_scripts();
26 }
```

[https://www.youtube.com/watch?v=h\\_41BPfhDt0](https://www.youtube.com/watch?v=h_41BPfhDt0)

If you're running on the latest TrinityCore revision, then that's all that you need to do. Earlier revisions required you to also add your file to the cmake lists, but that's no longer required ever since the hotswap implementation that changed things a bit.

Starter's Development Guide  
Vol. I