**BUNGIE C++ GUIDELINES & RAZORS**

Aug 31, 2021 - Destiny Dev Team

There's a lot of teamwork and ingenuity that goes into making a game like Destiny. We have talented people across all disciplines working together to make the best game that we can. However, achieving the level of coordination needed to make Destiny isn’t easy.   
  
It's like giving a bunch of people paintbrushes but only one canvas to share between them and expecting a high-quality portrait at the end. In order to make something that isn't pure chaos, some ground rules need to be agreed upon. Like deciding on the color palette, what size brushes to use in what situations, or what the heck you’re trying to paint in the first place. Getting that alignment amongst a team is incredibly important.   
  
One of the ways that we achieve that alignment over in engineering land is through coding guidelines: rules that our engineers follow to help keep the codebase maintainable. Today, I'm going to share how we decide what guidelines we should have, and how they help address the challenges we face in a large studio.   
  
The focus of this post will be on the game development side of things, using the C++ programming language, but even if you don't know C++ or aren't an engineer, I think you'll still find it interesting.

**WHAT'S A CODING GUIDELINE?**

A coding guideline is a rule that our engineers follow while they're writing code. They're commonly used to mandate a particular format style, to ensure proper usage of a system, and to prevent common issues from occurring. A well-written guideline is clearly actionable in its wording, along the lines of "Do X" or "Don't do Y" and explains the rationale for its inclusion as a guideline. To demonstrate, here’s a couple examples from our C++ guidelines:

**DON'T USE THE STATIC KEYWORD DIRECTLY**

*The "static" keyword performs a bunch of different jobs in C++, including declaring incredibly dangerous static function-local variables.   
  
You should use the more specific wrapper keywords in cseries\_declarations.h, such as static\_global, static\_local, etc. This allows us to audit dangerous static function-locals efficiently.*

**BRACES ON THEIR OWN LINES**

*Braces are always placed on a line by themselves.   
  
There is an exception permitted for single-line inline function definitions.*

Notice how there’s an exception called out in that second guideline? Guidelines are expected to be followed most of the time, but there's always room to go against one if it results in better code. The reasoning for that exception must be compelling though, such as producing objectively clearer code or sidestepping a particular system edge case that can't otherwise be worked around. If it’s a common occurrence, and the situation for it is well-defined, then we’ll add it as an official exception within the guideline.   
  
To further ground the qualities of a guideline, let’s look at an example of one from everyday life. In the USA, the most common rule you follow when driving is to drive on the right side of the road. You're pretty much always doing that. But on a small country road where there's light traffic, you'll likely find a dashed road divider that indicates that you're allowed to move onto the left side of the road to pass a slow-moving car. An exception to the rule. (Check with your state/county/city to see if passing is right for you. Please do not take driving advice from a tech blog post.)   
  
Now, even if you have a lot of well-written, thought-out guidelines, how do you make sure people follow them? At Bungie, our primary tool for enforcing our guidelines is through code reviews. A code review is where you show your code change to fellow engineers, and they’ll provide feedback on it before you share it with the rest of the team. Kind of like how this post was reviewed by other people to spot grammar mistakes or funky sentences I’d written before it was shared with all of you. Code reviews are great for maintaining guideline compliance, spreading knowledge of a system, and giving reviewers/reviewees the opportunity to spot bugs before they happen, making them indispensable for the health of the codebase and team.   
  
You can also have a tool check and potentially auto-fix your code for any easily identifiable guideline violations, usually for ones around formatting or proper usage of the programming language. We don't have this setup for our C++ codebase yet unfortunately, since we have some special markup that we use for type reflection and metadata annotation that the tool can't understand out-of-the-box, but we're working on it!   
  
Ok, that pretty much sums up the mechanics of writing and working with guidelines. But we haven't covered the most important part yet: making sure that guidelines provide value to the team and codebase. So how do we go about figuring out what's valuable? Well, let's first look at some of the challenges that can make development difficult and then go from there.

**CHALLENGES, YOU SAY?**

The first challenge is the programming language that we’re using for game development: C++. This is a powerful high-performance language that straddles the line between modern concepts and old school principles. It’s one of the most common choices for AAA game development to pack the most computations in the smallest amount of time. That performance is mainly achieved by giving developers more control over low-level resources that they need to manually manage. All of this (great) power means that engineers need to take (great) responsibility, to make sure resources are managed correctly and arcane parts of the language are handled appropriately.   
  
Our codebase is also fairly large now, at about 5.1 million lines of C++ code for the game solution. Some of that is freshly written code, like the code to support Cross Play in Destiny. Some of it is 20 years old, such as the code to check gamepad button presses. Some of it is platform-specific to support all the environments we ship on. And some of it is cruft that needs to be deleted. Changes to long-standing guidelines can introduce inconsistency between old and new code (unless we can pay the cost of global fixup), so we need to balance any guideline changes we want to make against the weight of the code that already exists.   
  
Not only do we have all of that code, but we're working on multiple versions of that code in parallel! For example, the development branch for Season of the Splicer is called v520, and the one for our latest Season content is called v530. v600 is where major changes are taking place to support The Witch Queen, our next major expansion. Changes made in v520 automatically integrate into the downstream branches, to v530 and then onto v600, so that the developers in those branches are working against the most up-to-date version of those files. This integration process can cause issues, though, when the same code location is modified in multiple branches and a conflict needs to be manually resolved. Or worse, something merges cleanly but causes a logic change that introduces a bug. Our guidelines need to have practices that help reduce the odds of these issues occurring.   
  
Finally, Bungie is a large company; much larger than a couple college students hacking away at games in a dorm room back in 1991. We're 150+ engineers strong at this point, with about 75 regularly working on the C++ game client. Each one is a smart, hardworking individual, with their own experiences and perspectives to share. That diversity is a major strength of ours, and we need to take full advantage of it by making sure code written by each person is accessible and clear to everyone else.   
  
Now that we know the challenges that we face, we can derive a set of principles to focus our guidelines on tackling them. At Bungie, we call those principles our C++ Coding Guideline Razors.

**RAZORS? LIKE FOR SHAVING?**

Well, yes. But no. The idea behind the term [razor](https://en.wikipedia.org/wiki/Philosophical_razor) here is that you use them to "shave off" complexity and provide a sharp focus for your goals (addressing the challenges we went through above). Any guidelines that we author are expected to align with one or more of these razors, and ones that don't are either harmful or just not worth the mental overhead for the team to follow.   
  
I'll walk you through each of the razors that Bungie has arrived at and explain the rationale behind each one, along with a few example guidelines that support the razor.

**#1 FAVOR UNDERSTANDABILITY AT THE EXPENSE OF TIME-TO-WRITE**

*Every line of code will be read many times by many people of varying backgrounds for every time an expert edits it, so prefer explicit-but-verbose to concise-but-implicit.*

When we make changes to the codebase, most of the time we're taking time to understand the surrounding systems to make sure our change fits well within them before we write new code or make a modification. The author of the surrounding code could've been a teammate, a former coworker, or you from three years ago, but you've lost all the context you originally had. No matter who it was, it's a better productivity aid to all the future readers for the code to be clear and explanative when it was originally written, even if that means it takes a little longer to type things out or find the right words.   
  
Some Bungie guidelines that support this razor are:

* snake\_case as our naming convention.
* Avoiding abbreviation (eg
* Encouraging the addition of helpful inline comments.

Below is a snippet from some of our UI code to demonstrate these guidelines in action. Even without seeing the surrounding code, you can probably get a sense of what it's trying to do.

Code

**#2 AVOID DISTINCTION WITHOUT DIFFERENCE**

*When possible without loss of generality, reduce mental tax by proscribing redundant and arbitrary alternatives.*

This razor and the following razor go hand in hand; they both deal with our ability to spot differences. You can write a particular behavior in code multiple ways, and sometimes the difference between them is unimportant. When that happens, we'd rather remove the potential for that difference from the codebase so that readers don't need to recognize it. It costs brain power to map multiple things to the same concept, so by eliminating these unnecessary differences we can streamline the reader's ability to pick up code patterns and mentally process the code at a glance.   
  
An infamous example of this is "tabs vs. spaces" for indentation. It doesn't really matter which you choose at the end of the day, but a choice needs to be made to avoid code with mixed formatting, which can quickly become unreadable.   
  
Some Bungie coding guidelines that support this razor are:

* Use American English spelling (ex "color" instead of "colour").
* Use post increment in general usage (
* Miscellaneous whitespace rules and high-level code organization within a file.

**#3 LEVERAGE VISUAL CONSISTENCY**

*Use visually-distinct patterns to convey complexity and signpost hazards*

The opposite hand of the previous razor, where now we want differences that indicate an important concept to really stand out. This aids code readers while they're debugging to see things worth their consideration when identifying issues.   
  
Here's an example of when we want something to be really noticeable. In C++ we can use the preprocessor to remove sections of code from being compiled based on whether we're building an internal-only version of the game or not. We'll typically have a lot of debug utilities embedded in the game that are unnecessary when we ship, so those will be removed when we compile for retail. We want to make sure that code meant to be shipped doesn’t accidentally get marked as internal-only though, otherwise we could get bugs that only manifest in a retail environment. Those aren't very fun to deal with.   
  
We mitigate this by making the C++ preprocessor directives really obvious. We use all-uppercase names for our defined switches, and left align all our preprocessor commands to make them standout against the flow of the rest of the code. Here's some example code of how that looks:

Code

Some Bungie coding guidelines that support this razor are:

* Braces should always be on their own line, clearly denoting nested logic.
* Uppercase for preprocessor symbols (eg
* No space left of the assignment operator, to distinguish from comparisons (eg
* Leverage pointer operators (

**#4 AVOID MISLEADING ABSTRACTIONS.**

*When hiding complexity, signpost characteristics that are important for the customer to understand.*

We use abstractions all the time to reduce complexity when communicating concepts. Instead of saying, "I want a dish with two slices of bread on top of each other with some slices of ham and cheese between them", you're much more likely to say, "I want a ham and cheese sandwich". A sandwich is an abstraction for a common kind of food.  
  
Naturally we use abstractions extensively in code. Functions wrap a set of instructions with a name, parameters, and an output, to be easily reused in multiple places in the codebase. Operators allow us to perform work in a concise readable way. Classes will bundle data and functionality together into a modular unit. Abstractions are why we have programming languages today instead of creating applications using only raw machine opcodes.  
  
An abstraction can be misleading at times though. If you ask someone for a sandwich, there's a chance you could get a hot dog back or a quesadilla depending on how the person interprets what a sandwich is. Abstractions in code can similarly be abused leading to confusion. For example, operators on classes can be overridden and associated with any functionality, but do you think it'd be clear that

* Use standardized prefixes on variables and types for quick recognition.
  + eg:
  + eg:
* No operator overloading for non-standard functionality.
* Function names should have obvious implications.
  + eg:
  + eg:
  + eg:

**#5 FAVOR PATTERNS THAT MAKE CODE MORE ROBUST.**

*It’s desirable to reduce the odds that a future change (or a conflicting change in another branch) introduces a non-obvious bug and to facilitate finding bugs, because we spend far more time extending and debugging than implementing.*

Just write perfectly logical code and then no bugs will happen. Easy right? Well... no, not really. A lot of the challenges we talked about earlier make it really likely for a bug to occur, and sometimes something just gets overlooked during development. Mistakes happen and that's ok. Thankfully there's a few ways that we can encourage code to be authored to reduce the chance that a bug will be introduced.  
  
One way is to increase the amount of state validation that happens at runtime, making sure that an engineer's assumptions about how a system behaves hold true. At Bungie, we like to use asserts to do that. An assert is a function that simply checks that a particular condition is true, and if it isn't then the game crashes in a controlled manner. That crash can be debugged immediately at an engineer’s workstation, or uploaded to our TicketTrack system with the assert description, function callstack, and the dump file for investigation later. Most asserts are also stripped out in the retail version of the game, since internal game usage and QA testing will have validated that the asserts aren't hit, meaning that the retail game will not need to pay the performance cost of that validation.  
  
Another way is to put in place practices that can reduce the potential wake a code change will have. For example, one of our C++ guidelines is to only allow a single

* Initialize variables at declaration time.
* Follow const correctness principles for class interfaces.
* Single
* Leverage asserts to validate state.
* Avoid native arrays and use our own containers.

**#6 CENTRALIZE LIFECYCLE MANAGEMENT.**

*Distributing lifecycle management across systems with different policies makes it difficult to reason about correctness when composing systems and behaviors. Instead, leverage the shared toolbox and idioms and avoid managing your own lifecycle whenever possible.*

When this razor is talking about lifecycle management, the main thing it's talking about is the allocation of memory within the game. One of the double-edged swords of C++ is that the management of that memory is largely left up to the engineer. This means we can develop allocation and usage strategies that are most effective for us, but it also means that we take on all of the bug risk. Improper memory usage can lead to bugs that reproduce intermittently and in non-obvious ways, and those are a real bear to track down and fix.   
  
Instead of each engineer needing to come up with their own way of managing memory for their system, we have a bunch of tools we've already written that can be used as a drop-in solution. Not only are they battle tested and stable, they include tracking capabilities so that we can see the entire memory usage of our application and identify problematic allocations.   
  
Some Bungie coding guidelines that support this razor are:

* Use engine-specified allocation patterns.
* Do not allocate memory directly from the operating system.
* Avoid using the Standard Template Library for game code.

**RECAP PLEASE**

Alright, let's review. Guideline razors help us evaluate our guidelines to ensure that they help us address the challenges we face when writing code at scale. Our razors are:

* Favor understandability at the expense of time-to-write
* Avoid distinction without difference
* Leverage visual consistency
* Avoid misleading abstractions
* Favor patterns that make code more robust
* Centralize lifecycle management

Also, you may have noticed that the wording of the razors doesn't talk about any C++ specifics, and that’s intentional. What's great about these is that they're primarily focused on establishing a general philosophy around producing maintainable code. They're mostly applicable to other languages and frameworks, whereas the guidelines that are generated from them are specific to the target language, project, and team culture. If you're an engineer, you may find them useful when evaluating the guidelines for your next project.

**WHO GUIDES THE GUIDELINES?**

Speaking of evaluation, who's responsible at Bungie for evaluating our guidelines? That would be our own C++ Coding Guidelines Committee. It's the committee's job to add, modify, or delete guidelines as new code patterns and language features develop. We have four people on the committee to debate and discuss changes on a regular basis, with a majority vote needed to enact a change.   
  
The committee also acts as a lightning rod for debate. Writing code can be a very personal experience with subjective opinions based on stylistic expression or strategic practices, and this can lead to a fair amount of controversy over what's best for the codebase. Rather than have the entire engineering org debating amongst themselves, and losing time and energy because of it, requests are sent to the committee where the members there can review, debate, and champion them in a focused manner with an authoritative conclusion.   
  
Of course, it can be hard for even four people to agree on something, and that’s why the razors are so important: they give the members of the committee a common reference for what makes a guideline valuable while evaluating those requests.