Preshing on Programming

- Twitter
- RSS

Navigate…

- Blog
- Archives
- About
- Contact

Mar 15, 2021

How C++ Resolves a Function Call

C is a simple language. You’re only allowed to have one function with each name. C++, on the other hand, gives you much more flexibility:

- You can have multiple functions with the same name (overloading).
- You can overload built-in operators like + and ==.
- You can write function templates.
- Namespaces help you avoid naming conflicts.

I like these C++ features. With these features, you can make `str1 + str2` return the concatenation of two strings. You can have a pair of 2D points, and another pair of 3D points, and overload `dot(a, b)` to work with either type. You can have a bunch of array-like classes and write a single `sort` function template that works with all of them.

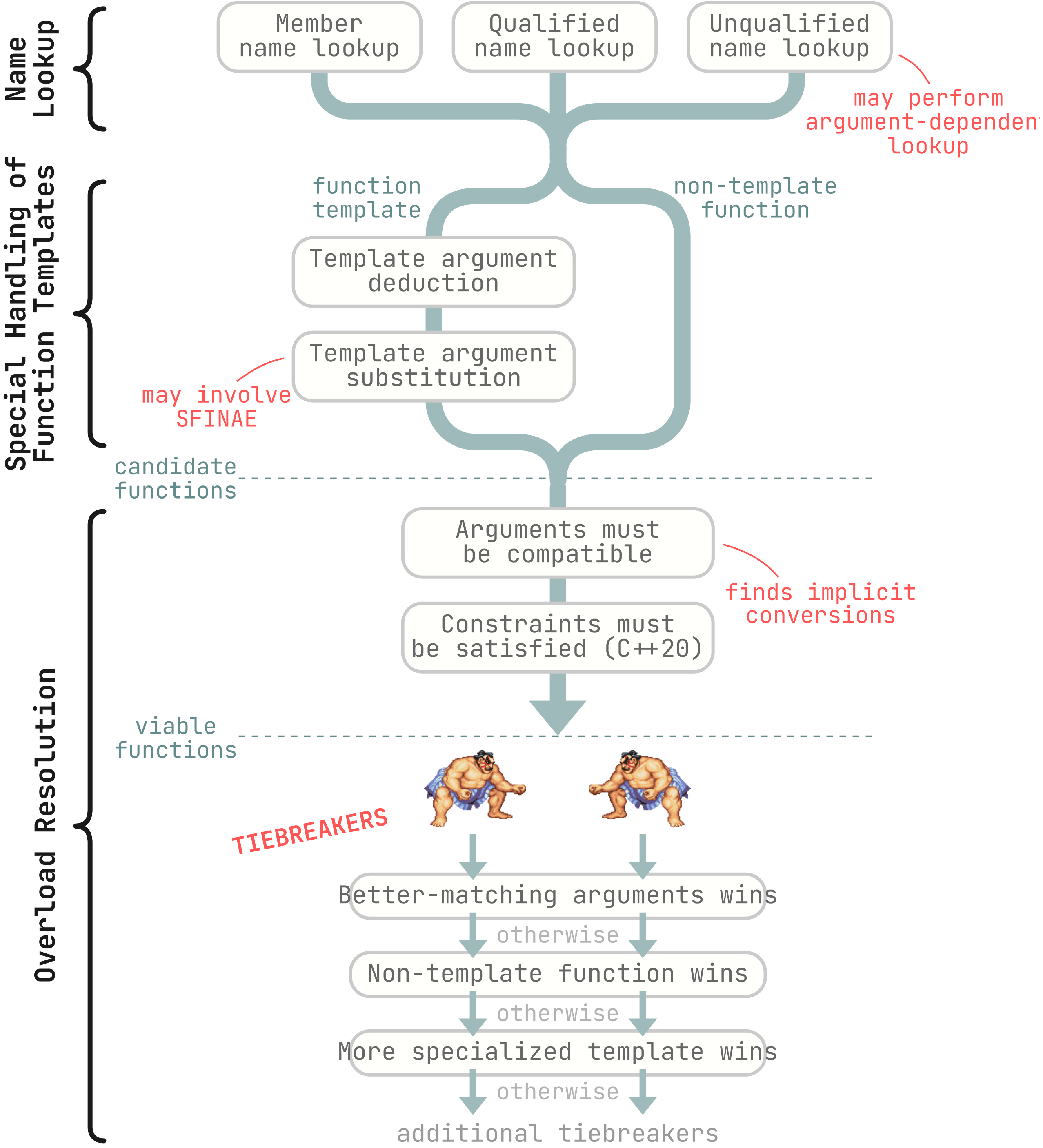
But when you take advantage of these features, it’s easy to push things too far. At some point, the compiler might unexpectedly reject your code with errors like:

```
error C2666: 'String::operator ==': 2 overloads have similar conversions
note: could be 'bool String::operator ==(const String &) const'
note: or       'built-in C++ operator==(const char *, const char *)'
note: while trying to match the argument list '(const String, const char *)'
```

Like many C++ programmers, I’ve struggled with such errors throughout my career. Each time it happened, I would usually scratch my head, search online for a better understanding, then change the code until it compiled. But more recently, while developing a new runtime library for Plywood, I was thwarted by such errors over and over again. It became clear that despite all my previous experience with C++, something was missing from my understanding and I didn’t know what it was.

Fortunately, it’s now 2021 and information about C++ is more comprehensive than ever. Thanks especially to cppreference.com, I now know what was missing from my understanding: a clear picture of the **hidden algorithm** that runs for every function call at compile time.

This is how the compiler, given a function call expression, figures out exactly which function to call:



Name Lookup

Member name lookup

Qualified name lookup

Unqualified name lookup

may perform argument-dependen lookup

Special Handling of Function Templates

function template

non-template function

Template argument deduction

Template argument substitution

may involve SFINAE

candidate functions

Overload Resolution

Arguments must be compatible

finds implicit conversions

Constraints must be satisfied (C++20)

viable functions

TIEBREAKERS

Better-matching arguments wins

otherwise

Non-template function wins

otherwise

More specialized template wins

otherwise

additional tiebreakers

These steps are enshrined in the C++ standard. Every C++ compiler must follow them, and the whole thing happens **at compile time** for every function call expression evaluated by the program. In hindsight, it’s obvious there has to be an algorithm like this. It’s the only way C++ can support all the above-mentioned features at the same time. This is what you get when you combine those features together.

I imagine the overall intent of the algorithm is to “do what the programmer expects”, and to some extent, it’s successful at that. You can get pretty far ignoring the algorithm altogether. But when you start to use multiple C++ features, as you might when developing a library, it’s better to know the rules.

So let’s walk through the algorithm from beginning to end. A lot of what we’ll cover will be familiar to experienced C++ programmers. Nonetheless, I think it can be quite eye-opening to see how all the steps fit together. (At least it was for me.) We’ll touch on several advanced C++ subtopics along the way, like argument-dependent lookup and SFINAE, but we won’t dive too deeply into any particular subtopic. That way, even if you know nothing else about a subtopic, you’ll at least know how it fits into C++’s overall strategy for resolving function calls at compile time. I’d argue that’s the most important thing.

Name Lookup

Our journey begins with a function call expression. Take, for example, the expression `blast(ast, 100)` in the code listing below. This expression is clearly meant to call a function named `blast`. But which one?

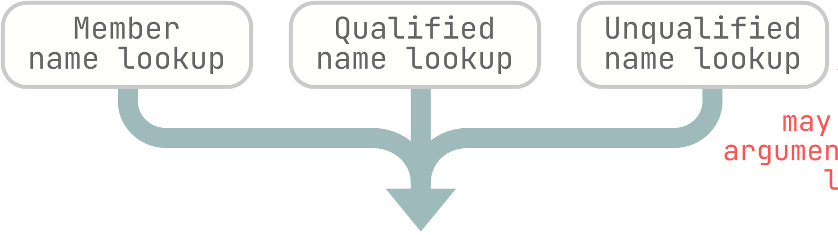
```
namespace galaxy {
    struct Asteroid {
        float radius = 12;
    };
    void blast(Asteroid* ast, float force);
}

struct Target {
    galaxy::Asteroid* ast;
    Target(galaxy::Asteroid* ast) : ast{ast} {}
    operator galaxy::Asteroid*() const { return ast; }
};

bool blast(Target target);
template <typename T> void blast(T* obj, float force);

void play(galaxy::Asteroid* ast) {
    blast(ast, 100);
}
```

The first step toward answering this question is **name lookup**. In this step, the compiler looks at all functions and function templates that have been declared up to this point and identifies the ones that *could be* referred to by the given name.

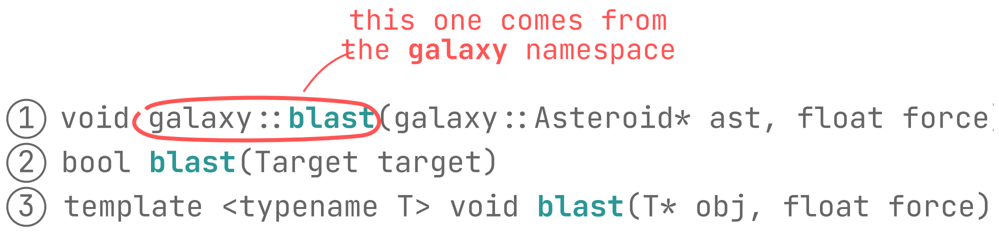


may perform argument-dependen lookup

As the flowchart suggests, there are three main types of name lookup, each with its own set of rules.

- **Member name lookup** occurs when a name is to the right of a . or -> token, as in foo->bar. This type of lookup is used to locate class members.
- **Qualified name lookup** occurs when a name has a :: token in it, like std::sort. This type of name is explicit. The part to the right of the :: token is only looked up in the scope identified by the left part.
- **Unqualified name lookup** is neither of those. When the compiler sees an unqualified name, like blast, it looks for matching declarations in various scopes depending on the context. There's a detailed set of rules that determine exactly where the compiler should look.

In our case, we have an unqualified name. Now, when name lookup is performed for a function call expression, the compiler may find multiple declarations. Let's call these declarations **candidates**. In the example above, the compiler finds three candidates:

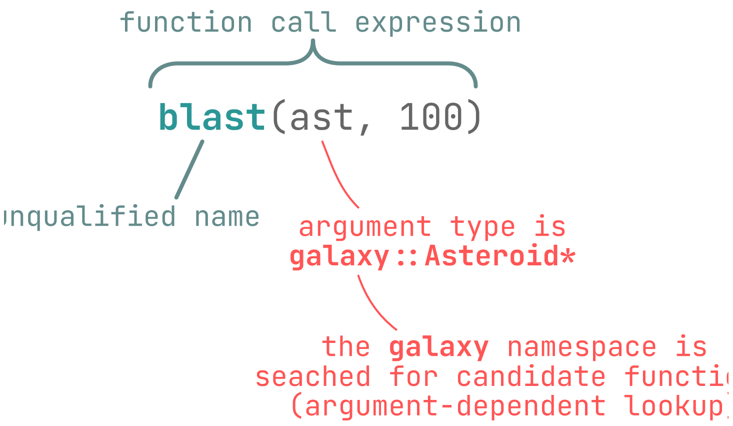


this one comes from
the **galaxy** namespace

```
① void galaxy::blast(galaxy::Asteroid* ast, float force
② bool blast(Target target)
③ template <typename T> void blast(T* obj, float force)
```

The first candidate, circled above, deserves extra attention because it demonstrates a feature of C++ that's easy to overlook: **argument-dependent lookup**, or **ADL** for short. I'll admit, I spent most of my C++ career unaware of ADL's role in name lookup. Here's a quick summary in case you're in the same boat. Normally, you wouldn't expect this function to be a candidate for this particular call, since it was declared inside the galaxy namespace and the call comes from *outside* the galaxy namespace. There's no using namespace galaxy directive in the code to make this function visible, either. So why is this function a candidate?

The reason is because any time you use an unqualified name in a function call – and the name doesn't refer to a class member, among other things – ADL kicks in, and name lookup becomes more greedy. Specifically, in addition to the usual places, the compiler looks for candidate functions *in the namespaces of the argument types* – hence the name "argument-dependent lookup".

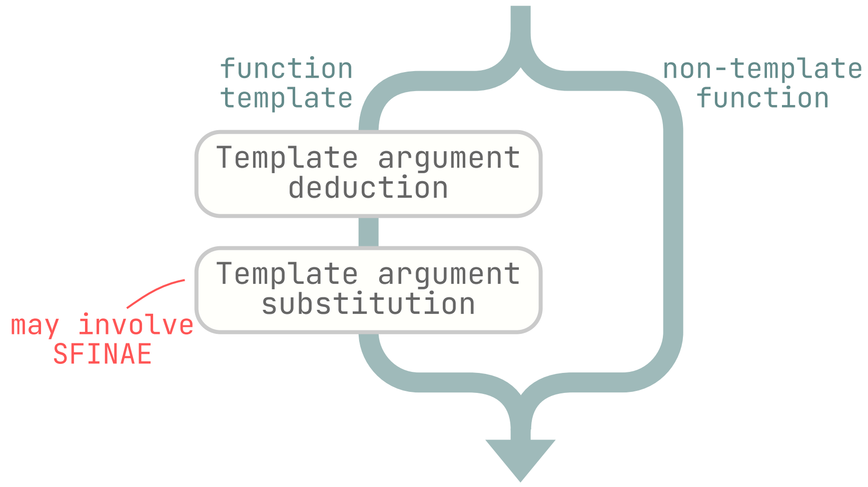


The complete set of rules governing ADL is more nuanced than what I've described here, but the key thing is that ADL only works with unqualified names. For qualified names, which are looked up in a single scope, there's no point. ADL also works when overloading built-in operators like + and ==, which lets you take advantage of it when writing, say, a math library.

Interestingly, there are cases where member name lookup can find candidates that unqualified name lookup can't. See this post by Eli Bendersky for details about that.

Special Handling of Function Templates

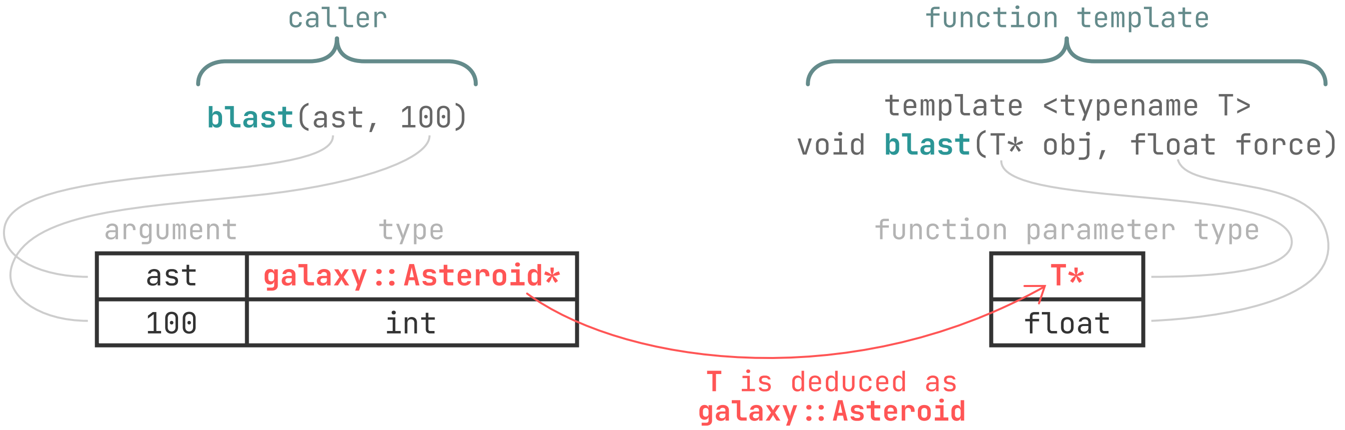
Some of the candidates found by name lookup are functions; others are function *templates*. There’s just one problem with function templates: You can’t call them. You can only call functions. Therefore, after name lookup, the compiler goes through the list of candidates and tries to turn each function template into a function.



In the example we’ve been following, one of the candidates is indeed a function template:

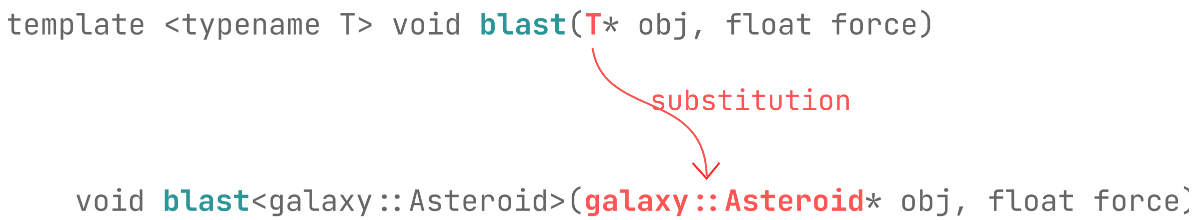
```
③ template <typename T> void blast(T* obj, float force
```

This function template has a single template parameter, `T`. As such, it expects a single template argument. The caller, `blast(ast, 100)`, didn’t specify any template arguments, so in order to turn this function template into a function, the compiler has to figure out the type of `T`. That’s where **template argument deduction** comes in. In this step, the compiler compares the types of the *function arguments* passed by the caller (on the left in the diagram below) to the types of the *function parameters* expected by the function template (on the right). If any unspecified template arguments are referenced on the right, like T, the compiler tries to deduce them using information on the left.



In this case, the compiler deduces T as galaxy::Asteroid because doing so makes the first function parameter T* compatible with the argument ast. The rules governing template argument deduction are a big subject in and of themselves, but in simple examples like this one, they usually do what you'd expect. If template argument deduction doesn't work – in other words, if the compiler is unable to deduce template arguments in a way that makes the function parameters compatible with the caller's arguments – then the function template is removed from the list of candidates.

Any function templates in the list of candidates that survive up to this point are subject to the next step: **template argument substitution**. In this step, the compiler takes the function template declaration and replaces every occurrence of each template parameter with its corresponding template argument. In our example, the template parameter T is replaced with its deduced template argument galaxy::Asteroid. When this step succeeds, we finally have the signature of a real function that can be called - not just a function template!



```
template <typename T> void blast(T* obj, float force)

   void blast<galaxy::Asteroid>(galaxy::Asteroid* obj, float force)
```

Of course, there are cases when template argument substitution can fail. Suppose for a moment that the same function template accepted a third argument, as follows:

```
template <typename T> void blast(T* obj, float force, typename T::Units mass = 5000);
```

If this was the case, the compiler would try to replace the T in T::Units with galaxy::Asteroid. The resulting type specifier, galaxy::Asteroid::Units, would be ill-formed because the struct galaxy::Asteroid doesn't actually have a member named Units. Therefore, template argument substitution would fail.

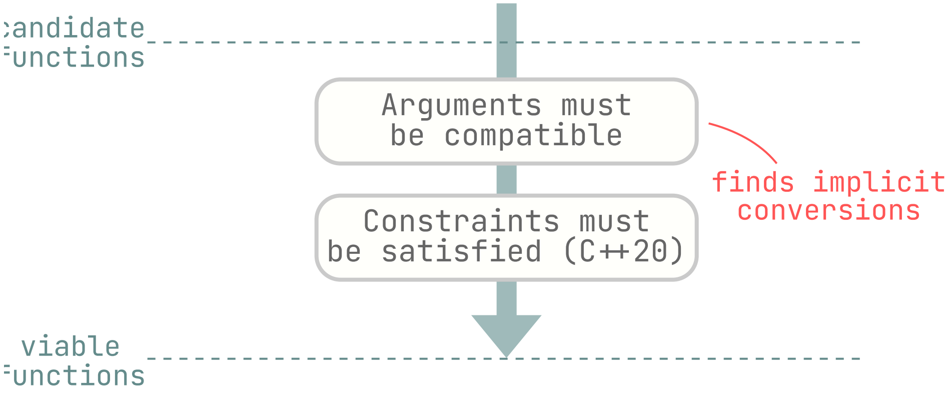
When template argument substitution fails, the function template is simply removed from the list of candidates - and at some point in C++'s history, people realized that this was a feature they could exploit! The discovery led to a entire set of metaprogramming techniques in its own right, collectively referred to as **SFINAE** (substitution failure is not an error). SFINAE is a complicated, unwieldy subject that I'll just say two things about here. First, it's essentially a way to rig the function call resolution process into choosing the candidate you want. Second, it will probably fall out of favor over time as programmers increasingly turn to modern C++ metaprogramming techniques that achieve the same thing, like constraints and constexpr if.

Overload Resolution

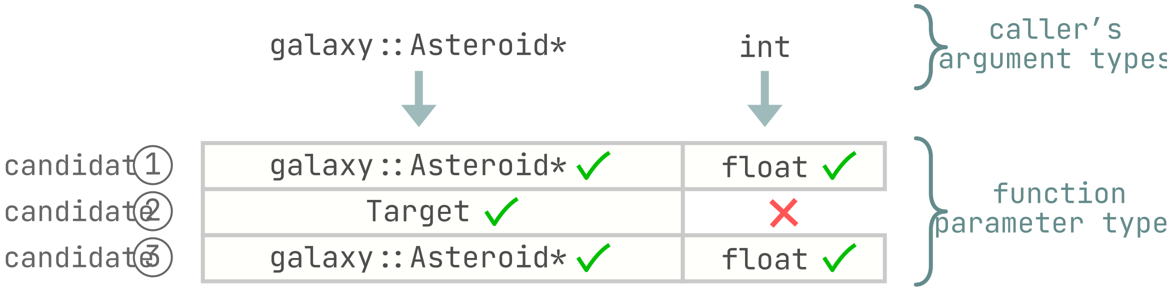
At this stage, all of the function templates found during name lookup are gone, and we're left with a nice, tidy set of **candidate functions.** This is also referred to as the **overload set.** Here's the updated list of candidate functions for our example:

```
① void galaxy::blast(galaxy::Asteroid* ast, float force)
② bool blast(Target target)
③ void blast<galaxy::Asteroid>(galaxy::Asteroid* obj, float force
```

The next two steps narrow down this list even further by determining which of the candidate functions are **viable** – in other words, which ones *could* handle the function call.



Perhaps the most obvious requirement is that the **arguments must be compatible**; that is to say, a viable function should be able to accept the caller's arguments. If the caller's argument types don't match the function's parameter types exactly, it should at least be possible to **implicitly convert** each argument to its corresponding parameter type. Let’s look at each of our example’s candidate functions to see if its parameters are compatible:



candidate ①	galaxy::Asteroid* ✓	float ✓
candidate ②	Target ✓	✗
candidate ③	galaxy::Asteroid* ✓	float ✓

Candidate 1

The caller’s first argument type galaxy::Asteroid* is an exact match. The caller’s second argument type int is implicitly convertible to the second function parameter type float, since int to float is a standard conversion. Therefore, candidate 1’s parameters are compatible.

Candidate 2

The caller’s first argument type galaxy::Asteroid* is implicitly convertible to the first function parameter type Target because Target has a converting constructor that accepts arguments of type galaxy::Asteroid*. (Incidentally, these types are also convertible in the other direction, since Target has a user-defined conversion function back to galaxy::Asteroid*.) However, the caller passed two arguments, and candidate 2 only accepts one. Therefore, candidate 2 is **not** viable.

```
① void galaxy::blast(galaxy::Asteroid* ast, float force)
~~② bool blast(Target target)~~
③ void blast<galaxy::Asteroid>(galaxy::Asteroid* obj, float force
```

Candidate 3

Candidate 3’s parameter types are identical to candidate 1’s, so it’s compatible too.

Like everything else in this process, the rules that control implicit conversion are an entire subject on their own. The most noteworthy rule is that you can avoid letting constructors and conversion operators participate in implicit conversion by marking them `explicit`.

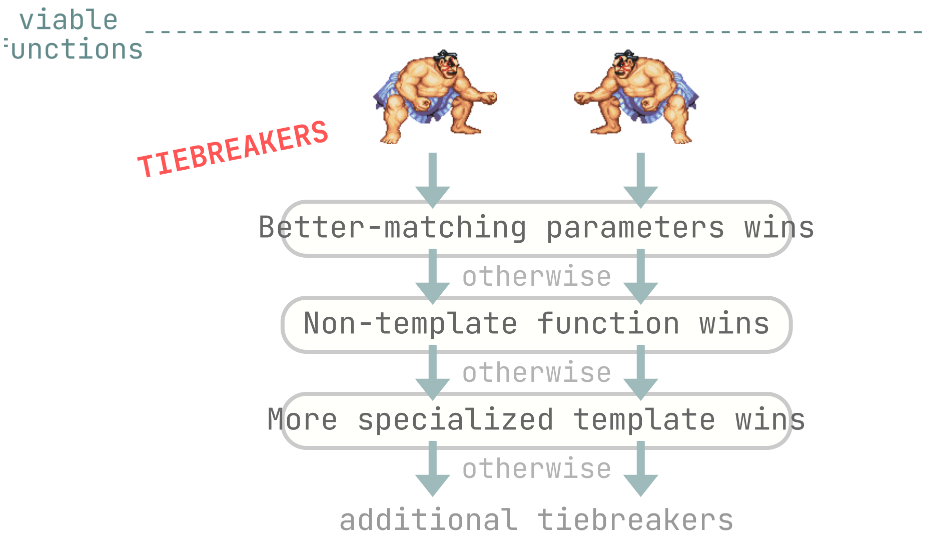
After using the caller’s arguments to filter out incompatible candidates, the compiler proceeds to check whether each function’s **constraints** are satisfied, if there are any. Constraints are a new feature in C++20. They let you use custom logic to eliminate candidate functions (coming from a class template or function template) without having to resort to SFINAE. They’re also supposed to give you better error messages. Our example doesn’t use constraints, so we can skip this step. (Technically, the standard says that constraints are also checked earlier, during template argument deduction, but I skipped over that detail. Checking in both places helps ensure the best possible error message is shown.)

Tiebreakers

At this point in our example, we’re down to two **viable** functions. Either of them could handle the original function call just fine:

```
① void galaxy::blast(galaxy::Asteroid* ast, float force)
② void blast<galaxy::Asteroid>(galaxy::Asteroid* obj, float force
```

Indeed, if either of the above functions was the only viable one, it *would* be the one that handles the function call. But because there are two, the compiler must now do what it always does when there are multiple viable functions: It must determine which one is the **best viable function**. To be the best viable function, one of them must “win” against every other viable function as decided by a sequence of tiebreaker rules.


viable
functions
TIEBREAKERS
Better-matching parameters wins
otherwise
Non-template function wins
otherwise
More specialized template wins
otherwise
additional tiebreakers

Let’s look at the first three tiebreaker rules.

First tiebreaker: Better-matching parameters wins

> C++ places the most importance on how well the caller’s argument types match the function’s parameter types. Loosely speaking, it prefers functions that require fewer implicit conversions from the given arguments. When both functions require conversions, some conversions are considered “better” than others. This is the rule that decides whether to call the `const` or non-`const` version of `std::vector`’s `operator[]`, for example.

In the example we've been following, the two viable functions have identical parameter types, so neither is better than the other. It's a tie. As such, we move on to the second tiebreaker.

Second tiebreaker: Non-template function wins

If the first tiebreaker doesn't settle it, then C++ prefers to call non-template functions over template functions. This is the rule that decides the winner in our example; viable function 1 is a non-template function while viable function 2 came from a template. Therefore, our **best viable function** is the one that came from the `galaxy` namespace:

```
void galaxy::blast(galaxy::Asteroid* ast, float force)
```

It's worth reiterating that the previous two tiebreakers are ordered in the way I've described. In other words, if there was a viable function whose parameters matched the given arguments better than all other viable functions, it would win *even if it was a template function*.

Third tiebreaker: More specialized template wins

In our example, the best viable function was already found, but if it wasn't, we would move on to the third tiebreaker. In this tiebreaker, C++ prefers to call "more specialized" template functions over "less specialized" ones. For example, consider the following two function templates:

```
template <typename T> void blast(T obj, float force);
template <typename T> void blast(T* obj, float force);
```

When template argument deduction is performed for these two function templates, the first function template accepts any type as its first argument, but the second function template accepts only pointer types. Therefore, the second function template is said to be **more specialized.** If these two function templates were the only results of name lookup for our call to `blast(ast, 100)`, and both resulted in viable functions, the current tiebreaker rule would cause the second one to be picked over the first one. The rules to decide which function template is more specialized than another are yet another big subject.

Even though it’s considered more specialized, it’s important to understand that the second function template isn’t actually a partial specialization of the first function template. On the contrary, they’re two completely separate function templates that happen to share the same name. In other words, they’re **overloaded.** C++ doesn’t allow partial specialization of function templates.

There are several more tiebreakers in addition to the ones listed here. For example, if both the spaceship <=> operator and an overloaded comparison operator such as `>` are viable, C++ prefers the comparison operator. And if the candidates are user-defined conversion functions, there are other rules that take higher priority than the ones I’ve shown. Nonetheless, I believe the three tiebreakers I’ve shown are the most important to remember.

Needless to say, if the compiler checks every tiebreaker and doesn’t find a single, unambiguous winner, compilation fails with an error message similar to the one shown near the beginning of this post.

After the Function Call Is Resolved

We’ve reached the end of our journey. The compiler now knows exactly which function should be called by the expression `blast(ast, 100)`. In many cases, though, the compiler has more work to do after resolving a function call:

- If the function being called is a class member, the compiler must check that member’s access specifiers to see if it’s accessible to the caller.
- If the function being called is a template function, the compiler attempts to instantiate that template function, provided its definition is visible.
- If the function being called is a virtual function, the compiler generates special machine instructions so that the correct override will be called at runtime.

None of those things apply to our example. Besides, they’re outside the scope of this post.

This post didn’t contain any new information. It was basically a condensed explanation of an algorithm already described by cppreference.com, which, in turn, is a condensed version of the C++ standard. However, the goal of this post was to convey the main steps without getting dragged down into details. Let’s take a look back to see just how much detail was skipped. It’s actually kind of remarkable:

- There’s an entire set of rules for unqualified name lookup.
- Within that, there’s a set of rules for argument-dependent lookup.
- Member name lookup has its own rules, too.
- There’s a set of rules for template argument deduction.

- There’s an entire family of metaprogramming techniques based on SFINAE.
- There’s a set of rules governing how implicit conversions work.
- Constraints (and concepts) are a completely new feature in C++20.
- There’s a set of rules to determine which implicit conversions are better than others.
- There’s a set of rules to determine which function template is more specialized than another.

Yeah, C++ is complicated. If you’d like to spend more time exploring these details, Stephan T. Lavavej produced a very watchable series of videos on Channel 9 back in 2012. Check out the first three in particular. (Thanks to Stephan for reviewing an early draft of this post.)

Now that I’ve learned exactly how C++ resolves a function call, I feel more competent as a library developer. Compilation errors are more obvious. I can better justify API design decisions. I even managed to distill a small set of tips and tricks out of the rules. But that’s a subject for another post.

« Flap Hero Code Review

Comments (8)

Ihar · *42 weeks ago*

Great Article. What soft/tool/approach did you used for creating such good diagrams/drawings ?

Reply ▾ 1 reply · *active 34 weeks ago*

↳ Jeff Preshing · *42 weeks ago*

Thanks! The diagrams were made in Inkscape.

Reply

cplusplus.es · *42 weeks ago*

Great article and very good explanation. Thanks for sharing with the community!

Reply

ilian · *42 weeks ago*

Great article! Kudos for including E.Honda! Also would be great if you add some fn() call conventions also.

Reply

dbjdbj · *42 weeks ago*

Was ADL invented so that iostreams user-defined streaming operators are possible?

Reply

Mario · *40 weeks ago*

Thank you for the article, it's very clear with beautiful drawings.

Reply

Maksadbek · *38 weeks ago*

The first example code can't be linked Clang 12: https://godbolt.org/z/evP4sq8EP Am I doing anything wrong ?

Reply ▾ 1 reply · *active 37 weeks ago*

↳ Jeff Preshing · *37 weeks ago*

You're not doing anything wrong. It's just not a complete program, since the definition of galaxy::blast isn't given.

Reply

Post a new comment

Enter text right here!

Comment as a Guest, or login: intensedebate WordPress.com twitter

Name

Displayed next to your comments.

Email

Not displayed publicly.

Subscribe to None

Submit Comment

Check out **Plywood**, a cross-platform, open source C++ framework:

Plywood

Recent Posts

- How C++ Resolves a Function Call
- Flap Hero Code Review
- A Small Open Source Game In C++
- Automatically Detecting Text Encodings in C++
- I/O in Plywood
- A New Cross-Platform Open Source C++ Framework
- A Flexible Reflection System in C++: Part 2
- A Flexible Reflection System in C++: Part 1
- How to Write Your Own C++ Game Engine

- Can Reordering of Release/Acquire Operations Introduce Deadlock?
- Here's a Standalone Cairo DLL for Windows
- Learn CMake's Scripting Language in 15 Minutes
- How to Build a CMake-Based Project
- Using Quiescent States to Reclaim Memory
- Leapfrog Probing
- A Resizable Concurrent Map
- New Concurrent Hash Maps for C++
- You Can Do Any Kind of Atomic Read-Modify-Write Operation
- Safe Bitfields in C++
- Semaphores are Surprisingly Versatile