11.6 — Inline functions

ALEX JUNE 17, 2021

The use of functions provides many benefits, including:

- The code inside the function can be reused.
- It is much easier to change or update the code in a function (which needs to be done once) than for every in-place instance. Duplicate code is a recipe for inefficiency and errors.
- It makes your code easier to read and understand, as you do not have to know how a function is implemented to understand what it does (assuming responsible function naming or comments).
- Functions provide type checking to ensure function call arguments match the function parameters (function-like macros don’t do this, which can lead to errors).
- Functions make your program easier to debug.

However, one major downside of functions is that every time a function is called, there is a certain amount of performance overhead that occurs. This is because the CPU must store the address of the current instruction it is executing (so it knows where to return to later) along with other registers, all the function parameters must be created and assigned values, and the program has to branch to a new location. Code written in-place is significantly faster.

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This can result in a substantial performance penalty.

C++ offers a way to combine the advantages of functions with the speed of code written in-place: inline functions. The **inline** keyword is used to request that the compiler treat your function as an inline function. When the compiler compiles your code, all inline functions are expanded in-place -- that is, the function call is replaced with a copy of the contents of the function itself, which removes the function call overhead! The downside is that because the inline function is expanded in-place for *every* function call, this can make your compiled code quite a bit larger, especially if the inline function is long and/or there are many calls to the inline function.

Consider the following snippet:

```cpp
int min(int x, int y)
{
    return x > y ? y : x;
}

int main()
{
    std::cout << min(5, 6) << '\n';
    std::cout << min(3, 2) << '\n';
    return 0;
}
```

This program calls function min() twice, incurring the function call overhead penalty twice. Because min() is such a short function, it is the perfect candidate for inlining:

```cpp
inline int min(int x, int y)
{
    return x > y ? y : x;
}
```

Now when the program compiles main(), it will create machine code as if main() had been written like this:

```cpp
int main()
{
    std::cout << (5 > 6 ? 6 : 5) << '\n';
    std::cout << (3 > 2 ? 2 : 3) << '\n';
    return 0;
}
```

This will execute quite a bit faster, at the cost of the compiled code being slightly larger.

Because of the potential for code bloat, inlining a function is best suited to short functions (e.g. no more than a few lines) that are typically called inside loops and do not branch. Also note that the inline keyword is only a recommendation -- the compiler is free to ignore your request to inline a function. This is likely to be the result if you try to inline a lengthy function!

Finally, modern compilers are now very good at inlining functions automatically -- better than humans in most cases. Even if you don't mark a function as inline, the compiler will inline functions that it believes will result in performance increases. Thus, in most cases, there isn't a specific need to use the inline keyword. Let the compiler handle inlining functions for you.

Best practice

Be aware of inline functions, but modern compilers should inline functions for you as appropriate, so there isn't a need to use the inline keyword in this context.

Inline functions are exempt from the one-definition per program rule

In previous chapters, we've noted that you should not implement functions (with external linkage) in header files, because when those headers are included into multiple .cpp files, the function definition will be copied into multiple .cpp files. These files will then be compiled, and the linker will throw an error because it will note that you've defined the same function more than once.

However, inline functions are exempt from the rule that you can only have one definition per program, because of the fact that inline functions do not actually result in a real function being compiled -- therefore, there's no conflict when the linker goes to link multiple files together.

This may seem like an uninteresting bit of trivia at this point, but next chapter we'll introduce a new type of function (a member function) that makes significant use of this point.

Even with inline functions, you generally should not define global functions in header files.

Next lesson
11.7 Function Pointers

Back to table of contents

Previous lesson
11.5 Returning values by value, reference, and address

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X