5.7 — Inline functions and variables

Consider the case where you need to write some code to perform some discrete task, like reading input from the user, or outputting something to a file, or calculating a particular value. When implementing this code, you essentially have two options:

1. Write the code as part of an existing function (called writing code “in-place” or “inline”).
2. Create a function (and possibly sub-functions) to handle the task.

Writing functions provides many potential benefits, as code in a function:

- Is easier to read and understand in the context of the overall program.
- Is easier to use, as you can call the function without understanding how it is implemented.
- Is easier to update, as the code in a function can be updated in one place.
- Is easier to reuse, as functions are naturally modular.

However, one downside of using a function is that every time a function is called, there is a certain amount of performance overhead that occurs. Consider the following example:

```
#include <iostream>

int min(int x, int y)
{
    return (x < y) ? x : y;
}

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

When a call to `min()` is encountered, the CPU must store the address of the current instruction it is executing (so it knows where to return to later) along with the values of various CPU registers (so they can be restored upon returning). Then parameters `x` and `y` must be instantiated and then initialized. Then the execution path has to jump to the code in the `min()` function. When the function ends, the program has to jump back to the location of the function call, and the return value has to be copied so it can be output. This has to be done for each function call.

For functions that are large and/or perform complex tasks, the overhead of the function call is typically insignificant compared to the amount of time the function takes to run. However, for small functions (such as `min()` above), the overhead costs can be larger than the time needed to actually execute the function's code! In cases where a small function is called often, using a function can result in a significant performance penalty over writing the same code in-place.

Inline expansion

Fortunately, the C++ compiler has a trick that it can use to avoid such overhead cost: **Inline expansion** is a process where a function call is replaced by the code from the called function's definition.

For example, if the compiler expanded the `min()` calls in the above example, the resulting code would look like this:

```
#include <iostream>

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

Note that the two calls to function `min()` have been replaced by the code in the body of the `min()` function (with the value of the arguments substituted for the parameters). This allows us to avoid the overhead of those calls, while preserving the results of the code.

The performance of inline code

Beyond removing the cost of function call, inline expansion can also allow the compiler to optimize the resulting code more efficiently -- for example, because the expression `((5 < 6) ? 5 : 6)` is now a constant expression, the compiler could further optimize the first statement in `main()` to `std::cout << 5 << '\n';`.

However, inline expansion has its own potential cost: if the body of the function being expanded takes more instructions than the function call being replaced, then each inline expansion will cause the executable to grow larger. Larger executables tend to be slower (due to not fitting as well in memory caches).

The decision about whether a function would benefit from being made inline (because removal of the function call overhead outweighs the cost of a larger executable) is not straightforward. Inline expansion could result in performance improvements, performance reductions, or no change to performance at all, depending on the relative cost of a function call, the size of the function, and what other optimizations can be performed.

Inline expansion is best suited to simple, short functions (e.g. no more than a few statements), especially cases where a single function call can be executed more than once (e.g. function calls inside a loop).

When inline expansion occurs

Every function falls into one of two categories, where calls to the function:

- May be expanded (most functions are in this category).
- Can't be expanded.

Most functions fall into the "may" category: their function calls can be expanded if and when it is beneficial to do so. For functions in this category, a modern compiler will assess each function and each function call to make a determination about whether that particular function call would benefit from inline expansion. A compiler might decide to expand none, some, or all of the function calls to a given function.

Tip

Modern optimizing compilers make the decision about when functions should be expanded inline.

The inline keyword, historically

Historically, compilers either didn't have the capability to determine whether inline expansion would be beneficial, or were not very good at it. For this reason, C++ provided the keyword `inline`, which was originally intended to be used as a hint to the compiler that a function would (probably) benefit from being expanded inline.

A function that is declared using the `inline` keyword is called an **inline function**.

Here's an example of using the `inline` keyword:

```
#include <iostream>

inline int min(int x, int y) // inline keyword means this function is an inline
function
{
    return (x < y) ? x : y;
}

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

However, in modern C++, the `inline` keyword is no longer used to request that a function be expanded inline. There are quite a few reasons for this:

- Using `inline` to request inline expansion is a form of premature optimization, and misuse could actually harm performance.
- The `inline` keyword is just a hint -- the compiler is completely free to ignore a request to inline a function. This is likely to be the result if you try to inline a lengthy function! The compiler is also free to perform inline expansion of functions that do not use the `inline` keyword as part of its normal set of optimizations.
- The `inline` keyword is defined at the wrong level of granularity. We use the `inline` keyword on a function definition, but inline expansion is actually determined per function call. It may be beneficial to expand some function calls and detrimental to expand others, and there is no syntax to influence this.

Modern optimizing compilers are typically good at determining which function calls should be made inline -- better than humans in most cases. As a result, the compiler will likely ignore or devalue any use of `inline` to request inline expansion for your functions.

Best practice

Do not use the `inline` keyword to request inline expansion for your functions.

The inline keyword, modernly

In previous chapters, we mentioned 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, which is a violation of the one-definition rule.

In modern C++, the term `inline` has evolved to mean "multiple definitions are allowed". Thus, an inline function is one that is allowed to be defined in multiple translation units (without violating the ODR).

Inline functions have two primary requirements:

- The compiler needs to be able to see the full definition of an inline function in each translation unit where the function is used (a forward declaration will not suffice on its own). The definition can occur after the point of use if a forward declaration is also provided. Only one such definition can occur per translation unit, otherwise a compilation error will occur.
- Every definition for an inline function must be identical, otherwise undefined behavior will result.

Rule

The compiler needs to be able to see the full definition of an inline function wherever it is used, and all such definitions must be identical (or undefined behavior will result).

The linker will consolidate all inline function definitions for an identifier into a single definition (thus still meeting the requirements of the one definition rule).

Here's an example:

main.cpp:

```
#include <iostream>

double circumference(double radius); // forward declaration

inline double pi() { return 3.14159; }

int main()
{
    std::cout << pi() << '\n';
    std::cout << circumference(2.0) << '\n';

    return 0;
}
```

math.cpp

```
inline double pi() { return 3.14159; }

double circumference(double radius)
{
    return 2.0 * pi() * radius;
}
```

Notice that both files have a definition for function `pi()` -- however, because this function has been marked as `inline`, this is acceptable, and the linker will de-duplicate them. If you remove the `inline` keyword from both definitions of `pi()`, you'll get an ODR violation (as duplicate definitions for non-inline functions are disallowed).

Inline functions are typically defined in header files, where they can be #included into the top of any code file that needs to see the full definition of the identifier. This ensures that all inline definitions for an identifier are identical.

pi.h:

```
#ifndef PI_H
#define PI_H

inline double pi() { return 3.14159; }

#endif
```

main.cpp:

```
#include "pi.h" // will include a copy of pi() here
#include <iostream>

double circumference(double radius); // forward declaration

int main()
{
    std::cout << pi() << '\n';
    std::cout << circumference(2.0) << '\n';

    return 0;
}
```

math.cpp

```
#include "pi.h" // will include a copy of pi() here

double circumference(double radius)
{
    return 2.0 * pi() * radius;
}
```

This is particularly useful for **header-only libraries**, which are one or more header files that implement some capability (no .cpp files are included). Header-only libraries are popular because there are no source files that need to be added to a project to use them and nothing that needs to be linked. You simply #include the header-only library and then can use it.

Related content

We show a practical example where inline is used for a random number generation header-only library in lesson 8.15 -- Global random numbers (Random.h).

For advanced readers

The following are implicitly inline:

- Functions defined inside a class, struct, or union type definition (14.3 -- Member functions).
- Constexpr / consteval functions (5.8 -- Constexpr and consteval functions).

- Functions implicitly instantiated from function templates (11.7 -- Function template instantiation).

For the most part, you should not mark your functions or variables as inline unless you are defining them in a header file (and they are not already implicitly inline).

Best practice

Avoid the use of the `inline` keyword unless you have a specific, compelling reason to do so (e.g. you're defining those functions or variables in a header file).

Why not make all functions inline and defined in a header file?

There are a few good reasons:

- It can increase your compile times. When a function in a code file changes, only that code file needs to be recompiled. When an inline function in a header file changes, every code file that includes that header (either directly or via another header) needs to recompiled. On large projects, this can have a drastic impact.
- It can lead to more naming collisions since you'll end up with more code in a single code file.

Inline variables C++17

In the above example, `pi()` was written as a function that returns a constant value. It would be more staightforward if `pi` were implemented as a (const) variable instead. However, prior to C++17, there were some obstacles and inefficiencies in doing so.

C++17 introduces **inline variables**, which are variables that are allowed to be defined in multiple files. Inline variables work similarly to inline functions, and have the same requirements (the compiler must be able to see an identical full definition everywhere the variable is used).

For advanced readers

The following are implicitly inline:

> Static constexpr data members 15.6 -- Static member variables.

Unlike constexpr functions, constexpr variables are not inline by default (excepting those noted above)!

We'll illustrate a common use for inline variables in lesson 7.9 -- Sharing global constants across multiple files (using inline variables), after we cover the prerequisite topic "global variables".