A deeper look at Inline functions

I think it's safe to say that all Overload readers know what C++ inline functions are. When we declare a function or member function as inline we are trying to avoid the overhead of a function call by getting the compiler to embed the code for the function at the point of the call. I'd like to take a few minutes and delve a bit deeper.

Inlines are always talked about in relation to performance. This isn't an article on performance but it is an article which talks about performance issues.

Optimisations

C++ inline functions present a potential optimisation to code : make a function *inline* and you save the overhead of a function call, that is: loading parameters onto the stack, making a jump, popping parameters off the stack, and then it all in reverse when you return.

Modern CPUs have pipelines and attempt to optimise code execution on the fly, a function call can disrupt this pipeline when it is encountered, and again, when it returns. So not only are we avoiding the function call but potentially we allow the CPU to further improve execution.

Not only this but a compiler may be able to optimise code which has been inlined in a way it cannot if there is a function call simply because the optimisation becomes visible, e.g. optimising register allocations. In the case of empty functions - fairly typical of constructors - making the function inline may allow the compiler to remove the call altogether.

Sometimes an inline function may save time and space. Making a function call requires instruction codes to make the call both on the part of the caller and callee. On occasions the number of bytes required for this may actually be greater than the number of bytes in the function body itself.

Inline as a better macro

C programmers used to simulate inline functions with macros. C++ inline functions where in part, an attempt to encode this as a language feature rather than yet another use of the macro pre-processor. However, they are not an exact replacement.

? If you use a macro the compiler has no desecration, the function will be inlined.

? Scope rules are different.

? The compiler will also ignore the inline if it considers the body of the function “too big.” How big is too big depends on your compiler.

? Inline code will not be generated when using a variable number of arguments to a function call (e.g. printf would not be inlined) or a variable sized data type, e.g.

```
void foo(int n) {
     char str[n];
     ....
}
```

? Compilers have a limit to the depth they will inline functions, e.g. if inline A calls inline B, which calls inline C and so on. This is often configurable through a pragma or command line switch.

Of course, every compiler will differ slightly and you I can’t guarantee that there isn’t a compiler out that can inline some of these things. Conversely, there are more reasons why your compiler will not inline a function that those given here.

These are clear cut cases but there are several other reasons why I avoid using inline functions. The main culprit is: code bloat, inline functions usually make your code bigger because you have duplicated code for each invocation of the function. Apart from needing a bigger hard disc code bloat has several implications:

? Increased number of CPU cache misses: a CPU may keep a frequently executed function entirely in it’s cache, if you increase the function size with lots of inline functions it may be too big for the cache, and because your functions are bigger fewer may be held in cache at once.

? Disc cache : if every function is inlined the OS may not load the entire program into memory and your disc will start to thrash.

So, don’t be surprised if making methods inline actually makes your program slower!

In addition embedded systems are very sensitive to code bloat: a PC may have a big CPU cache, oodles of fast RAM and a big fast hard disc but an embedded system probably won’t. In the extreme this may mean you need to fit your device with a bigger ROM, which means it costs more, which in a competitive market may make the difference between success and failure.

And some more catches....

Perhaps an more significant point is what a lot of inline functions says about your design. Inlining is typically used for *get* and *set* functions. Classes with lots of such functions aren't really hiding anything, they are just containers for values which begs the question, just how object oriented is your design?

Some other points that are worth noting about inlined functions are:

? Finding code in source files can be time consuming, if you have lots of inline functions you need to look in the .h files as well as the .cpp files. This may seem trivial but on a large system with several thousand files the time adds up.

? If some of your code is compiled with inlining enabled and some with them disabled you will encounter link errors.

? Some compilers have an *auto-inline* function which lets the compiler decide which functions to inline and which to leave as is. Of course, how aggressive this is, and hence how much difference it makes, is dependent on your compiler.

? Inlining is usually only enabled when optimised code is being generated, so your debug builds won't have any inlined functions.

? Although inline is commonly thought of as a C++ optimisation it is also available in C, and other languages, although these may need vendor specific extensions.

? Theoretically each inline function should have one, and only one implementation body. However, it would be possible to provide different bodies in different files. Stroustrup points out in the *Design & Evolution of C++* that most compilers don't check for this, we can hope that in the 7 years since he made his comments the situation has improved but I'm not sure. Before anyone think of a use for this "feature" consider the maintenance nightmare and please don't do it.

When to use inlines

Despite all this there are occasions when inlines make sense.

? Templates based code is always inlined : this is one of the reasons people believe templates lead to bigger executables.

? Sometimes making a function inline will improve performance for all the reasons given at the start.

The orthodox approach to performance today is to design a system for ease of maintenance, flexibility and extendibility, and only consider performance second. Usually, this means producing a working

something, measure it, and if necessary improve it. (Stroustrup questions if letting the compiler decide is the best policy.)

Conclusion

While inline functions may improve performance seldom will you be able to radically change a program's performance by making a bunch of functions inline. Inlines are not free: they will usually increase your program size and may actually impair performance.

They may also increase your development cycle because they increase coupling between files and lead to longer build times and more ripple effects.

Heavy reliance on inline functions may indicate flaws in your design; either a lack of object-orientedness or failure to design a system with the required performance characteristics.

Bibliography

Many books and web sites feature articles on optimisation techniques. While researching this piece I looked at quite a few specifically seeking out those that talked of inline functions. This is a selection of those I find interesting enough to read:

? Stroustrups views on inline functions are covered in the Design & Evolution of C++, Addison-Wesley 1994

? Pete Isensee has an interesting article on optimisation at www.tantalon.com/pete/cppopt/main.htm. In this he takes a slightly different view on when to optimise. He notes that Microsoft Visual C++ 6.0 only ever inlines the functions the developer specifies which makes the auto-inline options seems somewhat redundant! However my own experiments with the trivial code below suggest it will inline sometimes.

? Wind River Systems have a good article focusing on embedded systems at http://www.wrs.com/products/html/optimization_wp.html.

? There is an interesting article on optimisation in general at http://www2.ios.com/~jimwe19/program/optimize.htm, unfortunately beyond the e-mail address ihfhq@instructor.net I can't work out who actually wrote it!

these rules can be encoded in a good compiler it is hard for an individual to know and apply all these rules.

There are a few occasions where inline assembler is still useful though. Microsoft use a little bit in the ATL to "thunk" a function call, and sometimes hard coding a break point in your code (asm { int 03h; } in Intel speak) has it's uses.

Sidebar 2: Seeing inline at work

The most reliable way to see if a function is being inlined or not is to look at the output from the compiler. Most compilers have a switch to output assembler code for your inspection. In Visual C++ there are several /FA which control this – the "listing file type" in the project C++ settings box.

The other switches worth playing with are the /Ob which control the aggressiveness of inline expansion – the "Inline function expansion" option in C++ settings.

It's interesting to play around with these switches and this example to see what the compiler does:

```
struct Widget {
     int number_;
     Widget();
     int t() const;
};

Widget::Widget() : number_(99) { }
int Widget::t() const { return number_; }

int main()
{
     Widget w;
     int i = w.t();
     return i;
}
```