**Inline Functions in C++**

In C, one of the ways to preserve efficiency is through the use of macros, which allow you to make what looks like a function call without the normal function call overhead.

Problems with the use of preprocessor macros in C++:

1. A macro looks like a function call, but doesn’t always act like one. (True for C also)
2. The preprocessor has no permission to access class member data. This means preprocessor macro cannot be used as class member functions.

To retain the efficiency of the preprocessor macro, but to add the safety and class scoping of true functions, C++ has the inline function.

# Inline function

C++ implements the macro as inline function, which is a true function in every sense. Any behavior you expect from an ordinary function; you get from an inline function.

The only difference is that an inline function is expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated. Thus, you should (almost) never use macros, only inline functions.

**Why inline functions are used and what is the purpose of inline function?**

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). 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 overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline function to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

Any function defined within a class body is automatically inline, but you can also make a non-class function inline by preceding it with the inline keyword.

The syntax for defining the function inline is:

inline return-type function-name(parameters) {

// function code

}

Remember, **inlining is only a request to the compiler, not a command**. Compiler can ignore the request for inlining.

**Compiler may not perform inlining in such circumstances** like:

1. If a function contains a loop. (for, while, do-while)
2. If a function contains static variables.
3. If a function is recursive.
4. If a function return type is other than void, and the return statement doesn’t exist in function body.
5. If a function contains switch or goto statement.

## Inline function in header file

An inline function in a header file has a special status, since you must include the header file containing the function and its definition in every file where the function is used, but you don’t end up with multiple definition errors (however, the definition must be identical in all places where the inline function is included).

## Inline inside classes

To define an inline function, you must ordinarily precede the function definition with the inline keyword. However, this is not necessary inside a class definition.

class A {

public:

int default\_inline() {} // default inline

int make\_inline();

};

inline int A::make\_inline() {}; // making inline

## Advantages:

1. Function call overhead doesn’t occur.
2. It also saves the overhead of push/pop variables on the stack when function is called.
3. It also saves overhead of a return call from a function.
4. When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.
5. Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

## Disadvantages:

1. The added variables from the inlined function consumes additional registers, after in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.
2. If you use too many inline functions then the **size of the binary executable file will be large**, because of the duplication of same code.
3. Too much inlining can also **reduce your instruction cache hit rate**, thus **reducing the speed of instruction fetch from that of cache memory** to that of primary memory.
4. Inline function **may increase compile time overhead** if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.
5. Inline functions may not be useful for many embedded systems. Because **in embedded systems code size is more important than speed**.
6. Inline functions **might cause thrashing** because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

# Inline function and classes

It is also possible to define the inline function inside the class.

In fact, all the functions defined inside the class are implicitly inline.

Thus, all the restrictions of inline functions are also applied here.

If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

class S {

public:

inline int square(int s) { // redundant use of inline, bad use of inline

// this function is automatically inline

// function body

}

};

class S {

public:

int square(int s); // declare the function

};

inline int S::square(int s) { // use inline prefix, correct use of inline

/\* inline function \*/

}

# What is wrong with macro?

According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone.

Macro cannot access private members of class. Macros looks like function call but they are actually not.

class S {

int m;

public:

#define MAC(S::m) // CE: "::" may not appear in macro parameter list

};

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. Preprocessor macro is not capable for doing this.

One other thing is that the **macros are managed by preprocessor and inline functions are managed by C++ compiler**.

Virtual function cannot be inline

C++ compiler cannot perform inlining if the function is virtual. The **reason is call to a virtual function is resolved at runtime** instead of compile time. Virtual means wait until runtime **and inline means during compilation**, if the compiler doesn’t know which function will be called, how it can perform inlining?

One other thing to remember is that it is only **useful to make the function inline** **if the time spent during a function call is more compared to the function body execution time**.

An example where inline function has no effect at all:

inline void show() {

cout << "value of S = " << S << endl;

}

In general function which performs input output (I/O) operation shouldn’t be defined as inline because it spends a considerable amount of time.

Depending upon the compiler you are using the compiler may show you warning if the function is not expanded inline.

Programming languages like Java & C# doesn’t support inline functions.

But in Java, the compiler can perform inlining when the small final method is called, because final methods can’t be overridden by sub classes and call to a final method is resolved at compile time.

In C# JIT compiler can also optimize code by inlining small function calls (like replacing body of a small function when it is called in a loop).

Inline functions are the valuable feature of C++. An appropriate use of inline function can provide performance enhancement but if inline functions are used arbitrarily then they can’t provide better result.

In other words don’t expect better performance of program. Don’t make every function inline. It is better to keep inline functions as small as possible.

# Limitations

There are two situations in which the compiler cannot perform inlining.

1. If it must do this in multiple translation units (which would normally cause a multiple definition error), the linker is told to ignore the multiple definitions.
2. The compiler cannot perform inlining if the function is too complicated.

## Forward references

if an inline makes a forward reference to a function that hasn’t yet been declared in the class (whether that function is inline or not), it can seem like the compiler won’t be able to handle it.

#include <iostream>

using namespace std;

class Forward {

int i;

public:

Forward() : i(0) {}

// Call to undeclared function:

int f() const { return g() + 1; }

int g() const { return i; }

};

int main() {

Forward frwd;

frwd.f();

return 0;

}

# Reducing clutter

Dan Saks refers to member functions defined within classes using the Latin in situ (in place) and maintains that all definitions should be placed outside the class to keep the interface clean. Optimization, he argues, is a separate issue. If you want to optimize, use the inline keyword.

If you want to put the functions into documentation, it’s a simple cut-and-paste operation. In situ functions require more work and have greater potential for errors.

# Access functions

One of the most important uses of inlines inside classes is the access function. This is a small function that allows you to read or change part of the state of an object – that is, an internal variable or variables.

class Access {

int i;

public:

int read() const { return i; }

void set(int ii) { i = ii; }

};

## Accessors and mutators

Some people further divide the concept of access functions into accessors (to read state information from an object) and mutators (to change the state of an object). In addition, function overloading may be used to provide the same function name for both the accessor and mutator.

// Accessors & mutators

class Rectangle {

int wide, high;

public:

Rectangle(int w = 0, int h = 0) : wide(w), high(h) {}

int width() const { return wide; } // Read

void width(int w) { wide = w; } // Set

int height() const { return high; } // Read

void height(int h) { high = h; } // Set

};

# The inline keyword, modernly

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.

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 declaration, 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 affect this.

In modern C++, the inline concept has evolved to have a **new meaning: multiple definitions are allowed in the program**. This is true for functions as well as variables. Thus, if we mark a function as inline, then that function is allowed to have multiple definitions (in different files), as long as those definitions are identical.

# Constexpr functions are implicitly inline

Because constexpr functions may be evaluated at compile-time, the compiler must be able to see the full definition of the constexpr function at all points where the function is called.

This means that a constexpr function called in multiple files needs to have its definition included into each such file -- which would normally be a violation of the one-definition rule. To avoid such problems, constexpr functions are implicitly inline, which makes them exempt from the one-definition rule.

# References

<https://www.geeksforgeeks.org/c-plus-plus/>

<https://www.learncpp.com/>

<https://en.cppreference.com/w/cpp/language/functions>

# END