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.  
The syntax for defining the function inline is:

inline return-type function-name(parameters)

{

// function code

}

**Why to use –**  
In many places we create the functions for small work/functionality which contain simple and less number of executable instruction. Imagine their calling overhead each time they are being called by callers.  
When a normal function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function. Too much run time overhead.  
The C++ inline function provides an alternative. With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.  
With below pros, cons and performance analysis, you will be able to understand the “why” for inline keyword  
**Pros**: -   
1. It speeds up your program by avoiding function calling overhead.  
2. It saves overhead of variables push/pop on the stack, when function calling happens.  
3. It saves overhead of return call from a function.  
4. It increases locality of reference by utilizing instruction cache.  
5. By marking it as inline, you can put a function definition in a header file (i.e. it can be included in multiple compilation unit, without the linker complaining)  
  
**Cons:** -  
1. It increases the executable size due to code expansion.   
2. C++ inclining is resolved at compile time. Which means if you change the code of the inclined function, you would need to recompile all the code using it to make sure it will be updated  
3. When used in a header, it makes your header file larger with information which users don’t care.  
4. As mentioned above it increases the executable size, which may cause thrashing in memory. More number of page fault bringing down your program performance.  
5. Sometimes not useful for example in embedded system where large executable size is not preferred at all due to memory constraints.  
  
**When to use -**  
Function can be made as inline as per programmer need. Some useful recommendation is mentioned below-  
1. Use inline function when performance is needed.  
2. Use inline function over macros.  
3. Prefer to use inline keyword outside the class with the function definition to hide implementation details.  
  
**Key Points -**  
1. It’s just a suggestion not compulsion. Compiler may or may not inline the functions you marked as inline. It may also decide to inline functions not marked as inline at compilation or linking time.  
2. Inline works like a copy/paste controlled by the compiler, which is quite different from a pre-processor macro: The macro will be forcibly inclined, will pollute all the namespaces and code, won't be easy to debug.  
3. All the member function declared and defined within class are Inline by default. So, no need to define explicitly.  
4. Virtual methods are not supposed to be inclinable. Still, sometimes, when the compiler can know for sure the type of the object (i.e. the object was declared and constructed inside the same function body), even a virtual function will be inlined because the compiler knows exactly the type of the object.  
5. Template methods/functions are not always inclined (their presence in a header will not make them automatically inline).  
6. Most of the compiler would do in-lining for recursive functions but some compiler provides #pragmas-   
Microsoft c++ compiler - inline recursion(on) and once can also control its limit with inline depth.  
In gcc, you can also pass this in from the command-line with --max-inline-insns-recursive

**Inline functions provide following advantages over macros.**

* Since they are functions so type of arguments is checked by the compiler whether they are correct or not.
* There is no risk if called multiple times. But there is risk in macros which can be dangerous when the argument is an expression.
* They can include multiple lines of code without trailing backlashes.
* Inline functions have their own scope for variables and they can return a value.
* Debugging code is easy in case of Inline functions as compared to macros

**Inline Advantages**

* Inline substitution can pay off in several ways. First, it can eliminate the overhead in doing a function call. When a function is called, the following steps are usually taken:
* Argument values are copied to the stack or special registers.
* A return address is created and stored on the stack or to a register.
* The program branches to the function.
* A stack frame is set up for the local variables of the function.
* After the function finishes, the stack frame is torn down.
* The return address is retrieved.
* A branch is made to the return address.

Inline substitution allows the optimizer to do a better job. For example, the expression **10\*b** is a common sub-expression that occurs both in the original body of **caller1()** and the inline substituted code. After inclining, the optimizer can recognize that **10\*b** has the same value in both places and compute that expression only once and use it twice. Likewise, if the call **f(&c, 10)** was made, the compiler could perform the arithmetic in the assignment to **c** at compile time..

### Inline Disadvantages

The primary disadvantage to inline substitution is that it usually makes the program code bigger. In extreme cases, this can degrade program performance by increasing page faults and cache misses. Reading a page from the disk may take as long as executing hundreds of thousands of instructions. Poor cache performance may slow down a program by a factor of two. Reasonable care must be taken when inlining not to make the program so big that either paging or caching problems dominate the execution time.

There are also functions that it does not pay to inline. Consider:

if ((ptr = malloc(100)) == NULL)

die();

where the **die()** function prints an error message, performs a little cleanup, and then exits the program. There might be lots of calls to **die()**, **die()** might even be a very short function, but it would never pay to inline the function since it is never executed. Since the function calls are not executed, you want the calls to be as short as possible to minimize page faults and cache misses.

Comparison Chart

| **BASIS FOR COMPARISON** | **INLINE** | **MACRO** |
| --- | --- | --- |
| Basic | Inline functions are parsed by the compiler. | Macros are expanded by the preprocessor. |
| Syntax | inline return type funct\_name ( parameters ){ . . . } | #define macro\_name char\_sequence |
| Keywords Used | inline | #define |
| Defined | It can be defined inside or outside the class. | It is always defined at the start of the program. |
| Evaluation | It evaluates the argument only once. | It evaluates the argument each time it is used in the code. |
| Expansion | The compiler may not inline and expand all the functions. | Macros are always expanded. |
| Automation | The short functions, defined inside the class are automatically made onto inline functions. | Macros should be defined specifically. |
| Accessing | An inline member function can access the data members of the class. | Macros can never be the members of the class and can not access the data members of the class. |
| Termination | Definition of inline function terminates with the curly brackets at the end of the inline function. | Definition of macro terminates with the new line. |
| Debugging | Debugging is easy for an inline function as error checking is done during compilation. | Debugging becomes difficult for macros as error checking does not occur during compilation. |
| Binding | An inline function binds all the statements in the body of the function very well as the body of the function start and ends with the curly brackets. | A macro faces the binding problem if it has more than one statement, as it has no termination symbol. |

# **Macros vs Functions**

Macros are **pre-processed** which means that all the macros would be processed before your program compiles. However, functions are **not preprocessed but compiled**.

In macros, no type checking(incompatible operand, etc.) is done and thus use of micros can lead to errors/side-effects in some cases. However, this is not the case with functions. Also, macros do not check for compilation error (if any). Consider the following two codes:

#include<stdio.h>

#define CUBE(b) b\*b\*b

int main()

{

     printf("%d", CUBE(1+2));

     return 0;

}

#include<stdio.h>

int cube(int a)

{

     return a\*a\*a;

}

int main()

{

    printf("%d", cube(1+2));

    return 0;

}

**Conclusion:**  
Macros are no longer recommended as they cause following issues. There is a better way in modern compilers that is inline functions and const variable. Below are disadvantages of macros:

a) There is no type checking

b) Difficult to debug as they cause simple replacement.

c) Macro don’t have namespace, so a macro in one section of code can affect other section.

d) Macros can cause side effects as shown in above CUBE() example.