**Inline functions.**

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

Preceding a function declaration with the *inline* specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

For example, the *concatenate* function above may be declared inline as:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | inline string concatenate (const string& a, const string& b)  {  return a+b;  } |  |

This informs the compiler that when concatenate is called, the program prefers the function to be expanded inline, instead of performing a regular call. *inline* is only specified in the function declaration, not when it is called.

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the *inline* specifier. Therefore, this specifier merely indicates the compiler that *inline* is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11  12  13  14  15  16 | #include<iostream>  #include<string> //for string and getline keyword  using namespace std;  inline string concatenate(string& s1, string& s2)  {  return s1 + s2;  }  int main()  {  string s1, s2;  cout << "Enter first string value:" << endl;  getline(cin, s1);  cout << "Enter second string value:" << endl;  getline(cin, s2);  cout << "The concatenated string values:" << endl << concatenate(s1, s2) << endl;  } |

Explanation with instances:

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*

*}*

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | #include <iostream>  using namespace std;  inline int cube(int s)  {  return s \* s \* s;  }  int main()  {  cout << "The cube of 3 is: " << cube(3) << "\n";  return 0;  } | The cube of 3 is: 27 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

Interesting fact:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | #include <iostream>  using namespace std;  inline int cube(int s)  {  return s \* s \* s;  }  int main()  {  cout << "The cube of 3 is: " << cube << "\n";  return 0;  } | The cube of 3 is: 00B412F8 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

Calling function with just its name-identifier will print its function address.

However, if the *cube* function is called with parenthesis with no arguments it will be an error, where it will print too little amount of arguments in calling function:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | #include <iostream>  using namespace std;  inline int cube(int s)  {  return s \* s \* s;  }  int main()  {  cout << "The cube of 3 is: " << cube() << "\n";  return 0;  } | Error: too few arguments in a function call | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

Another case is that if we input more arguments than required in function parameter. This will output an error warning about too many arguments in a function call.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | #include <iostream>  using namespace std;  inline int cube(int s)  {  return s \* s \* s;  }  int main()  {  cout << "The cube of 3 is: " << cube(3, 5) << "\n";  return 0;  } | Error: too many arguments in a function call | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

Anyways, 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 functions provide following 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.

Inline function 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 registers 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.