7.5 — Inline functions

BY ALEX ON JULY 30TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

The use of functions provides many benefits, including:

- The code inside the function can be reused.
- It is much easier to change or update the code in a function (which needs to be done once) than for every in-place instance. Duplicate code is a recipe for inefficiency and errors.
- It makes your code easier to read and understand, as you do not have to know how a function is implemented to understand what it does (assuming responsible function naming or comments).
- Functions provide type checking to ensure function call arguments match the function parameters (function-like macros don't do this, which can lead to errors).
- Functions make your program easier to debug.

However, one major downside of functions is that every time a function is called, there is a certain amount of performance overhead that occurs. This is because the CPU must store the address of the current instruction it is executing (so it knows where to return to later) along with other registers, all the function parameters must be created and assigned values, and the program has to branch to a new location. Code written in-place is significantly faster.

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 can result in a substantial performance penalty.

C++ offers a way to combine the advantages of functions with the speed of code written in-place: inline functions. The **inline** keyword is used to request that the compiler treat your function as an inline function. When the compiler compiles your code, all inline functions are expanded in-place -- that is, the function call is replaced with a copy of the contents of the function itself, which removes the function call overhead! The downside is that because the inline function is expanded in-place for *every* function call, this can make your compiled code quite a bit larger, especially if the inline function is long and/or there are many calls to the inline function.

Consider the following snippet:

```
1  int min(int x, int y)
2  {
3      return x > y ? y : x;
4  }
5
6  int main()
7  {
8      std::cout << min(5, 6) << '\n';
9      std::cout << min(3, 2) << '\n';
10     return 0;
11 }
```

This program calls function min() twice, incurring the function call overhead penalty twice. Because min() is such a short function, it is the perfect candidate for inlining:

```
1  inline int min(int x, int y)
2  {
3      return x > y ? y : x;
4  }
```

Now when the program compiles main(), it will create machine code as if main() had been written like this:

```
1  int main()
```

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

This will execute quite a bit faster, at the cost of the compiled code being slightly larger.

Because of the potential for code bloat, inlining a function is best suited to short functions (e.g. no more than a few lines) that are typically called inside loops and do not branch. Also note that the inline keyword is only a recommendation -- the compiler is free to ignore your request to inline a function. This is likely to be the result if you try to inline a lengthy function!

Finally, modern compilers are now very good at inlining functions automatically -- better than humans in most cases. Even if you don't mark a function as inline, the compiler will inline functions that it believes will result in performance increases. Thus, in most cases, there isn't a specific need to use the inline keyword. Let the compiler handle inlining functions for you.

Rule: Be aware of inline functions, but modern compilers should inline functions for you as appropriate, so there isn't a need to use the keyword.

Inline functions are exempt from the one-definition per program rule

In previous chapters, we've noted 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.

However, inline functions are exempt from the rule that you can only have one definition per program, because of the fact that inline functions do not actually result in a real function being compiled -- therefore, there's no conflict when the linker goes to link multiple files together.

This may seem like an uninteresting bit of trivia at this point, but next chapter we'll introduce a new type of function (a member function) that makes significant use of this point.

Even with inline functions, you generally should not define global functions in header files.

 **7.6 -- Function overloading**

 **Index**

 **7.4a -- Returning values by value, reference, and address**

C++ TUTORIAL | PRINT THIS POST

76 comments to 7.5 — Inline functions

Samira Ferdi
August 26, 2019 at 6:28 pm · Reply

Hi, Alex and Nascardriver!

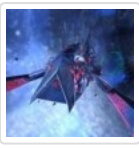
I wanna clarify my understanding.

> The inline keyword is used to request that the compiler treat your function as an inline function. When the compiler compiles your code, all inline functions are expanded in-place - that is, the function call is replaced with a copy of the contents of the function itself, which removes the function call overhead! The downside is that because the inline function is expanded in-place for every function call, this can make your compiled code quite a bit larger, especially if the inline function is long and/or there are many calls to the inline function.
>

So, it basically means, using inline function has consequences duplicate code (not in the sense of duplicate definition) because every inline function is called, the copy of its content are made, and this make the size of compiled code quite bigger. Am I right?

nascardriver
August 27, 2019 at 3:52 am · Reply

yes



potterman28wxcv
August 17, 2019 at 8:23 am · Reply

Hello!

> In previous chapters, we've noted 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.
>
> However, inline functions are exempt from the rule that you can only have one definition per program,

because of the fact that inline functions do not actually result in a real function being compiled -- therefore, there's no conflict when the linker goes to link multiple files together.

Is it still true for functions that the compiler chose not to inline? You mentioned earlier that "inline" was just an indication, and that the compiler did not have to inline - but then, what does the compiler do when compiling an "inline" function that he chose not to inline?

nascardriver
August 18, 2019 at 1:04 am · Reply

> Is it still true for functions that the compiler chose not to inline?
Yes. `inline` actually inlining the function is just a nice side effect, but not the main objective of `inline`. As long as all definitions of the `inline` function are identical, it is allowed to be defined multiple times.
I don't know how compilers deal with `inline` functions. I suppose they compile them once and check for equality whenever they are encountered again.

potterman28wxcv
August 22, 2019 at 4:32 am · Reply

After having tested to define the same inline functions two times, I doubt it is actually allowed by the compiler.

See this example: https://godbolt.org/z/k1lxMJ

nascardriver
August 22, 2019 at 4:43 am · Reply

The function is allowed to be defined multiple times, but not multiple times in the same translation unit (ie. file).
I don't think godbolt supports multiple files. Try with an offline compiler and copy an inline function to another .cpp file. That works.
Inline functions don't have to be definable twice in the same file, because header guards prevent that already.

potterman28wxcv
August 22, 2019 at 7:24 am · Reply

I figured it out - I think defining a function inline also implicitly makes it static.

The following configuration does not work:

main.cpp

```
1  void hello();
2
3  int main()
4  {
5      hello();
6      return 0;
7  }
```

inline.cpp

```
1  #include <iostream>
2
```

```
inline void hello(){
    std::cout << "Hello" << std::endl;
}
```

However, if now I define hello as just `void hello()`, it works correctly.

Alireza
May 20, 2019 at 4:03 am · Reply

Hello and greetings,
Which compilers are modern that use inline as appropriate ?

And one other question:
I wrote this code, but it gives me a warning:

```
// header file h.h
namespace display {
    inline void gettingChar();
}
```

```
 // contests of function h.cpp
namespace display {
     void gettingChar()
        {
            std::cin.clear(); // reset any error flags
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            std::cin.get(); // get one more char from the user
        }
}
```

```
 // main.cpp
#include <iostream>
#include "h.h"

int main()
{
    display::gettingChar();
    return 0;
}
```

```
C.h:21: warning: inline function 'void display::gettingChar()' used but never defined
     inline void gettingChar();
                 ^
```

What is wrong with this code ?

nascardriver
May 20, 2019 at 6:25 am · Reply

> it gives me a warning
inline functions cannot be defined in different files, unless you're defining it in the file that you're using the function in. eg. if you moved the definition of @gettingChar into @main.cpp, it should work. To make the function work no matter where it's used, define inline functions in the header.

Louis Cloete
February 9, 2019 at 1:19 pm · Reply

Alex, can you maybe consider to explain function-like macros here too?

I know you don't want to because it is a bad practice to use them, but the fact is that you might come across use of it in someone else's codebase. You can treat it like object-like macros, saying it is the wrong way even before you explain it.

If you still don't want to explain them, I will understand, but it is a shortcoming of the site I picked up.

If you touch them later and I just didn't see them yet, I apologise.

Alex
February 10, 2019 at 1:13 pm · Reply

Fair point. And there are a few uses of object-like macros that are valid, particularly for debugging. I'll add it to my to-do when I revamp this chapter.

Alireza
May 20, 2019 at 3:56 am · Reply

Hi,
I agree with Louis Cloete. Because you should explain properties and attributes of C++. Like all of properties, it is better that you'd tell us to avoid them (like fixed arrays and global variables(unless in such cases)).
I think it's better you to explain it(and after that tell to avoid it) than not to explain.

magaji::hussaini
December 21, 2018 at 2:40 am · Reply

Hi alex/nascardriver/others
I think inline functions are not actually like described, I tried these:

```
//Inline.....
inline void printMainsVars() {

        std::cout << static_cast<short>(length) << '\t' <<
                     static_cast<short>(width)  << '\n';
}
```

func-like macro

```
#define printMainsVars() (std::cout << static_cast<short>(length) << '\t' << static_cast<sho
```

In main:

```
int main() {

    char length{ 26 }, width{ 50 };
    printMainsVars();

  return 0;
}
```

But only the macro work as expected.
Inline version causes error of undeclared length & width
If it can get main's variables then inlines are not (directly) replace with their content codes...

Alex
December 24, 2018 at 12:04 pm · Reply

Inline functions work just like normal functions do, except the compiler optimizes out the function call when producing the compiled code. This means the compiler isn't doing a straight text-replace like the preprocessor would do, but is rather doing some internal magic to make it work.

saj
November 30, 2018 at 4:30 am · Reply

how do we forward declare an inline function, do we use "inline" keyword there too?

```
#include <limits>

inline void cinIgnore();

inline void cinIgnore()
{
cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
```

Alex
December 2, 2018 at 3:25 pm · Reply

Only the definition needs to be marked as inline. The forward declarations don't.

Louis Cloete
January 21, 2019 at 4:36 pm · Reply

@Alex, I think you should maybe mention this in the lesson. I also suspect that the forward declaration should explicitly not have the inline keyword in front of it. I used the inline keyword in front of some functions I wanted to use outside the .cpp file. When I forward-declared them as

```
inline type funcName();
```

in the header file, I got a linker error about undefined reference to 'funcName()' as well as a compiler warning about inline function 'type funcName()' used but never defined. It seems as if the compiler/linker treats a anything following the inline keyword as a function definition, even if you meant it as a forward declaration by putting a ; after the prototype, and then the linker can't find the definition of the inline function, so it errors. My compiler and IDE is MinGW g++ on Code::Blocks and Windows 7 32 bit.

EDIT: After some reading: the compiler can only inline functions defined in a header file or in the same .cpp file as the caller. The exception is when link-time optimisation is enabled. For a function to be able to safely live in a header, you must use the inline keyword, else the ODR will be violated See https://stackoverflow.com/a/12414229/10909834

Louis Cloete
January 21, 2019 at 5:41 pm · Reply

More interesting information on how the compiler inlines functions declared in different places: https://stackoverflow.com/a/11377991/10909834

EDIT: @Alex, it does not work for me to forward declare a function with the inline keyword in its definition as just the type and parameters. The only way that works for me to call an inline

function from another code file, is to include the actual function definition in the header file and then include the header into the code file where I want to use the inline function.

Louis Cloete
January 22, 2019 at 4:31 am · Reply

With relation to my edit above: here is the answer I got after further searching: https://stackoverflow.com/a/50666010/10909834 It would seem to be the case indeed that an inline function must be defined in the header file if you want to use it in multiple .cpp files. If it is not defined in the .h file, you won't be able to call it from another code file. The definition must be present in full in each file that uses it, else you will get a linker error about an undefined reference. That much is apparently enforced by the standard.

Alex
January 23, 2019 at 9:47 pm · Reply

Correct: You can use a forward declaration for an inline function, but the definition of the inline function must end up in the same file. This is because the inline function doesn't result in an actual function, so there's no way for the linker to connect a call to the function to the definition of that function in a separate file.

For what it's worth, on Visual Studio and Code::Blocks I tried forward declaring an inline function both with and without the inline keyword in the forward declaration. Both accepted either way, so I suspect the inline keyword is simply ignored on the forward declaration if present.

Louis Cloete
January 23, 2019 at 10:12 pm · Reply

Probably. The thing I tried, was to forward declare an inline function and then use it in a different file. That will definitely not work.

I deduced a set of rules for myself with inline functions:

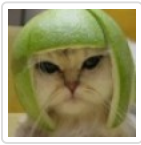
1. Since an inline function is permitted to have more than one identical definition, always define them in the header file, except if your program is so trivial that it just uses one .cpp file OR if you deliberately want the function to be accessible in only one file.

2. Always define a function used in more than one file as inline if you suspect that it could benefit from inlining. This will enable the compiler to inline it in more than one .cpp file, instead of being limited to only the .cpp file in which the function is defined. If the function will not benefit from inlining, the compiler will still be free to ignore the "inline" keyword.

The "inline" keyword should properly be understood as "allow this function to have multiple identical definitions" instead of "replace calls to this function with its body." It doesn't force the compiler to inline the function, but it enables you to define the function in the header, contrary to normal rules for functions, which helps the compiler to optimise potentially inlinable functions. This is because the compiler must see the body of the function to be able to inline it, and if it is inside a .cpp file, it can not be seen except for that one .cpp file, while

if it is in a header, it can be seen in all files that #include that header (and thus use the function).

3. If you enable Link-Time Optimisation for your compiler, you can declare possibly inlinable functions in a .cpp file and the compiler will still optimise them. It is a highly technical subject, and if you want to know more, you can read the links in my posts above. An interesting tidbit, though, is that GCC does this by consolidating all .cpp files into one large .cpp file before compiling, so that all functions in the program are available to the compiler at the same time. MSVC does this differently. The -flto flag enables Link-Time Optimisation in GCC/MinGW.



Alex
January 24, 2019 at 9:06 pm · Reply

1) Agreed.
2) I agree with part of this. The inline keyword does allow the function to have multiple identical definitions. But it's also a hint to the compiler that you'd like the function to be inlined. The compiler is free to ignore that request. As far as inline usage, if a function is small and performance critical, define it as inline. An inline function may be copied into each place it is CALLED, removing the overhead of making a function call. Inlining all your functions will probably result in massive code bloat.
3) Interesting! Thanks for sharing.

Louis Cloete
January 25, 2019 at 5:41 am · Reply

About 2): I meant that you should define as inline in the .h file short trivial functions such as:

```
1  bool isRow(char row)
2  {
3      return (row >= '1' && row <= '8');
4  }
```

if you want to use it across the program and want the compiler to be able to optimise throughout the program, not just in the .cpp file it was declared.

Anyway, if I understand correctly, the "inline" keyword is kind of deprecated with the advent of LTO anyway. If you want the best-optimised code your compiler can offer, enable LTO and don't bother with the "inline" keyword.

Chetan
March 20, 2018 at 11:07 pm · Reply

[code]what do you mean by "inline function is expanded in-place for every function call"?[/]

nascardriver
March 21, 2018 at 10:31 am · Reply

Hi Chetan!

When calling a normal function your computer will jump to that function and continue executing code in there until it's done, then jump back to where in came from.
Inline functions aren't actually functions. During compilation, every call to an inline function is replaced with the contents of the inline function.

```
#include <iostream>

inline void myFunction(void)
{
  std::cout << "Hello World!" << std::endl;
}

int main(void)
{
  myFunction();

  return 0;
}
```

Will be turned into

#include <iostream>

int main(void)
{
 std::cout << "Hello World!" << std::endl;

 return 0;
}

Ran
November 25, 2017 at 6:54 pm · Reply

Greeting:

Not understang the following code:

```
72 inline Foam::stringList& Foam::argList::args()
73 {
74     return args_;
75 }
```

It means that the combaination of the flowing codes?

```
72 inline Foam::stringList::args()
73 {
74     return args_;
75 }
```

```
72 inline Foam::argListargs()
73 {
74     return args_;
75 }
```

Correct me if I am wrong.

Alex
November 26, 2017 at 9:33 am · Reply

No. This is a member function of class argList which is nested inside class Form. The function returns a Foam::stringList, which is either another nested class or type alias.

Lim Che Ling
October 3, 2017 at 7:18 pm · Reply

"This may seem like an uninteresting bit of trivia at this point, but next chapter we'll introduce a new type of function (a member function) that makes significant use of this point." --> This leads to checking out on "8.2 — Classes and class members".

In 8.2's Member Functions, "Member functions can be defined inside or outside of the class definition."

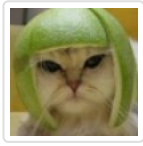
Could I clarify that when member functions is defined inside the class definition, it is considered 'inline' functions whereas 'non-inline' if outside of the class definition?
Does this go the same for constructor?

When I worked on Chapter12.1's Animal quiz, I have main.cpp, Animal.cpp, Animal.hpp where in Animal.hpp's class Animal's constructor can be defined inside the class or outside the class. (My Animal.cpp has only one line of code #include "Animal.hpp" by default).

When the Animal's constructor is inside the class definition, program is compiled perfectly. I tried to move out the constructor below the class definition to become:

```
Animal::Animal(std::string name, const char* speak):m_name(name), m_speak(speak){}
```

I get linker error. Is it because there are #include "Animal.hpp" in both main.cpp and Animal.cpp, so when constructor is moved out of class definition (not inline anymore), my project has breached one-definition rule?

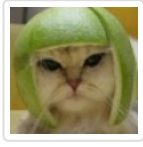


Alex
October 8, 2017 at 7:22 pm · Reply

All your assertions above are correct. I'm not sure why you're getting an error since you didn't tell me what error you're getting. If you move the constructor call out, you need to ensure it's not also still defined in the class, or you'll violate the one definition rule. I do cover how to do constructors outside the class definition in lesson 8.9.

nikos-13
July 8, 2017 at 6:08 am · Reply

"The function provides type checking."
What do you mean with that?



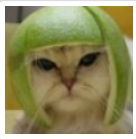
Alex
July 8, 2017 at 9:16 pm · Reply

I meant that the function ensures that type of the arguments and parameters match. As opposed to function-like macros, which don't do this.

Omri
May 10, 2017 at 6:27 pm · Reply

Hello Alex,
Regarding:

"That said, you still should generally not define global functions in..."
Perhaps should be "... inline functions..."?



Alex
May 11, 2017 at 2:33 pm · Reply

I added "(inline or otherwise)" to the text to denote I'm talking about both inline and non-inline functions in this case.

Sam
December 12, 2016 at 7:47 am · Reply

Hi!It's better to say "have got" than "have gotten" since there's no such word (at least in actual grammar). :)

Alex
December 12, 2016 at 11:09 am · Reply

I agree. Text updated. Thanks!

Nyap
June 1, 2016 at 3:32 am · Reply

Does the program get moved to memory when it's ran? I heard that happens on linux which is why you can delete a program and continue to use it until you close it but didn't know that happens everywhere

Alex
June 1, 2016 at 9:44 pm · Reply

I don't think modern OSes typically load the entire executable into memory at launch, just parts of it, with the rest loaded on demand. The OS handles all of loading/memory management stuff for you, so it's not typically exposed.

Chris
April 9, 2016 at 7:05 am · Reply

Is it possible to create a function pointer to an inline function such that every time the function pointer is called, a the function that it points to will be compiled into your code in place.

Alex
April 9, 2016 at 11:42 am · Reply

No, that is essentially what an inline function is in the first place.

malt
April 6, 2016 at 2:49 am · Reply

Little question:

inline functions == closures == lambda functions ?

Great tuto by the way!

Alex
April 7, 2016 at 12:08 pm · Reply

No, closures/lambda functions are a different feature from inline functions.

Michael
August 7, 2015 at 6:50 pm · Reply

Hello Alex!

Thanks again for your excellent tutorials!

I have a question about inline functions and multiple files.
I have a project that has main.cpp functions.cpp and header.h - to keep things organized.

I have this small function stored in functions.cpp:

```
1  char yourChoice(std::string cstMessage)
2  {
3  while (true){
4      cin.clear();
5      cout << endl;
6      cout << cstMessage;
7
8      system("stty raw");
9      char choice = 'n';
10     choice = getchar();
11     system("stty cooked");
12     return choice;
13     }
14 };
```

in header.h I have its declaration:

char yourChoice(std::string cstMessage); //simple char return function (no Enter key)

and I #include "header.h" in both main.cpp and functions.cpp

I call this function from main.cpp

Everything was working great until I decided to inline this function.

I get an error "main.cpp|87|undefined reference to `yourChoice(std::string)"

All I did was add 'inline' in both the header and function.cpp file, like this:

```
1  inline char yourChoice(std::string cstMessage) {...};
```

However, when I copied this function from functions.cpp to main.cpp and put it before my int main(){} and added that 'inline' as above - everything compiled smoothly just like before inlinig it.

Am I doing something wrong?

Alex
August 7, 2015 at 9:57 pm · Reply

An inline function needs to have a definition in every place it gets used. You got an error because inline yourChoice() wasn't defined in function main() where you were using it. This

wasn't a problem when it was non-inline because C++ can resolve the dependency, but inline functions have different rules.

c++ freak
June 7, 2015 at 10:12 pm · Reply

i think lot of things are missing here ... your lesson are making us to be more confused than we were.. it will be good if u post a program using inline function..
people are getting confused here regrading what type of functions when made inline are ignored by the compiler.. why does compiler ignores them ...
... please reply to above comments ..
anyways nice tutorials

Jacob Perkins
August 2, 2015 at 9:31 am · Reply

You're confusing two important terms: the c++ standard [vs] the c++ implementation. It is up to the c++ implementation to decide what should be inlined and what shouldn't. Various C++ compilers like gcc or the visual studio c++ compiler are just different implementations of the c++ standard. How different compilers treat your code will vary from one compiler to another because obviously they are not all built the same way.

Sasan
March 25, 2015 at 2:01 am · Reply

Does Inline functions effect/differ in the release version of program? (.exe for example)

Alex
October 27, 2015 at 11:56 am · Reply

Not to my knowledge. An inline function should work the same way in release or debug.
That said, your compiler may set different defaults for whether (or how aggressively) it inlines functions. So some functions may be inlined in the release build and not in the debug build.

Ashu
September 1, 2014 at 1:01 am · Reply

I have Little doubt regarding Inline function:

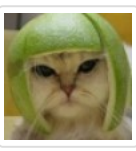
```
void main()
{
int i=2;
fun();
printf("%d",i);
}

inline void fun()
{
int i=4;
printf("%d", i);
}
```

As compiler make copy of this fun() function and insert inside main function body. why compiler not give error of variable int i re declaration.

Marc
June 15, 2015 at 11:39 am · Reply

I am pretty sure that the inline function has it's own scope and the compiler would automatically use the local i.



Alex
October 27, 2015 at 11:53 am · Reply

The compiler has to be smart about how it inlines code. As Marc suggests, it can put the function code in its own block to avoid naming collisions with already declared variables. It also needs to know how to convert parameters to arguments, which can be tricky since arguments passed by value and reference are handled differently.

dice3000
August 8, 2013 at 9:18 pm · Reply

I used to thing that this was done by default to all functions. Now I don't :)

Jay
December 24, 2009 at 9:34 am · Reply

Hi Alex,

One quick question. When a compiler is smart enough to decide whether "inline" is applicable or not (based on some hidden criteria by compiler as you explained above), as a common practice why can't we define each and every function as an inline function and leave it up to the compiler to decide whether to treat it as an inline or not. what could be the ups and downs by following such a style of coding?

Sajjad
April 29, 2010 at 3:43 am · Reply

i have the same question.

ellankavi
July 27, 2010 at 1:24 pm · Reply

Well, I was thinking if the compiler is smart enough, why can't it do this work implicitly... But, I too would like to know the answer to Jay's question...

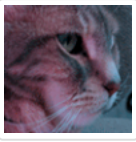
Joe
January 1, 2011 at 12:39 pm · Reply

I don't think compilers are smart enough to know whats best. It only knows very specific situations, where its definitely not efficient and will ignore inline under those conditions. but it certainly isn't smart enough to make every decision for you.

Alex
October 27, 2015 at 11:44 am · Reply

You could, but the compiler will not only ignore inline requests you've made that it doesn't think should be inlined, it will also inline functions you haven't marked as inline that it thinks should be inlined. So the inline keyword is just a hint that the compiler is free to ignore or add as it pleases.

As a result, there's generally no need to inline functions yourself. If you're optimizing your code, you could try inlining functions to see if there's a performance improvement, but outside of that and a few other obscure edge cases, you're better off not cluttering your code up with uses of the inline keyword.

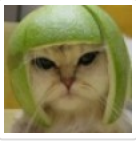


Peter Baum
May 9, 2018 at 2:36 pm · Reply

Could you expand upon this idea within the lesson. Specifically,

1. Is there a way to know whether or not a function has been inlined by the compiler?

2. Is there a way to force the compiler to inline or not inline specific code? Here I am thinking that the compiler may have no way of knowing how often specific code sections will execute. Also the compiler may not have information regarding the relative importance of a code section with respect to execution speed.



Alex
May 10, 2018 at 10:02 pm · Reply

1) Not that I am aware of.
2) Not that I am aware of. :) There may be hacky ways of doing this, but the language doesn't officially support forcing the compiler to inline/not inline.

nascardriver
May 11, 2018 at 4:05 am · Reply

1. By manual static analysis, yes. Dynamically at run-time, yes, but strongly compiler specific and not nice.
2. There's no standard way. msvc++ has @__forceinline and @noinline. I suggest you not to use them, but if you just want to play around, go ahead.

References
* __forceinline - https://msdn.microsoft.com/en-us/library/bw1hbe6y.aspx
* noinline - https://msdn.microsoft.com/en-us/library/kxybs02x.aspx

Peter Baum
May 12, 2018 at 6:24 am · Reply

Nice job nascardriver. Thanks.

DARK_BYTE
June 18, 2009 at 11:57 pm · Reply

Wow c++ is a lot more interesting and a lot harder than python I have to say!
:P

kamal
March 13, 2009 at 3:09 pm · Reply

Is it advisable to make this function as inline? (It has loop which iterates for large number of times)

```
void fun1()
{
 for(int i<0;i<10000;i++)
 {
   cout<<endl<<i;
 }
}

<!--formatted-->
```

Alex
May 1, 2009 at 7:17 pm · Reply

I wouldn't -- I only use inlines for functions that are absolutely trivial. Anything with a loop doesn't qualify. I'm pretty sure if you put a loop in an inline function the compiler will just treat it as a normal function.

pravin
September 18, 2009 at 5:16 am · Reply

how would i know whether for above (for loop) function, the compiler had accepted/rejected our request
of making the function inline..

the tutorials are very good..

Len
October 2, 2015 at 8:41 am · Reply

I wondered this as well. There is a Codeblocks compiler option "Warn if a function can not be inlined and it was declared as inline"

You waited 6 years for an answer, you finally got one...

HtLrR
February 2, 2009 at 1:03 pm · Reply

Thanks,that easy to understand ..:)

Daniel
June 23, 2008 at 11:24 pm · Reply

Wow! These tutorials are growing almost as fast as I can read them :)
Could you please explain the major differences between using an inline function and a #define macro?

Alex
June 26, 2008 at 8:41 am · Reply

Let me direct you to **this article**, which does a good job laying out the differences.

Personally, I never use #defines any more except for using in conjunction with #ifdef.

Daniel
June 27, 2008 at 11:56 pm · Reply

thanks, that explained a lot.

Zafer
February 4, 2008 at 10:57 am · Reply

What happens when the compiled code becomes large by using the inline keyword? Why is it disadvantageous to have a large compile code?

Alex
February 4, 2008 at 12:35 pm · Reply

Well, it takes up more space on disk, for one. It can also have a negative impact on how quickly your program executes due to the way the operating system pages stuff in and out of memory. Bloated code means more page loads, which adds overhead.

Zafer
February 4, 2008 at 1:27 pm · Reply

I don't know what a page load means. Is this a different type of overhead because normally inline functions are meant to reduce overhead as explained in the text.

Alex
February 4, 2008 at 4:36 pm · Reply

Page loading is actually an operating system concept and has nothing to do with C++ itself. It's overhead that comes from the operating system itself due to the way memory is set up.

Your computer has a certain amount of physical memory on it. This memory needs to hold all the programs that are running. However, it's sometimes the case that our programs want to use more memory than we have physical memory on the machine! In the old days, you'd probably get a memory error and your program wouldn't run until you freed some stuff up. However, most modern operating systems use a system called virtual memory. Virtual memory is basically a technique that lets the operating system assign more memory to programs than it actually physically has!

The way this works is all of the physical memory in the machine is divided into chunks called pages. When a program is loaded, the operating system load it into how ever many pages it needs. However, if we run out of free pages, the operating system will make room by temporarily writing out some older pages to disk. This is what a swap drive is for. Then, when those old pages become needed again, it reads them back into memory from disk (and possibly writes out something else)

If you've ever run Windows XP (or Vista) on 512 megs of ram (or less), you know how slow it can be. Why? Because the operating system is writing pieces of itself out to disk and reading them back in all the time.

Bloated programs tend to cause the operating system to have to read in and write out more pages (for a variety of reasons). Since writing pages to disk is extremely slow, this is something you want to avoid as much as possible.

Zafer
February 4, 2008 at 5:38 pm · Reply

Ok then can we say that the inline keyword speeds up the execution only when the function is not too large otherwise it can cause the program to be even slower?

Alex
February 4, 2008 at 7:05 pm · Reply

Abusing the inline keyword could, in theory, cause the program to be even slower than usual. However, there is no guarantee that this is strictly true, as there are other factors that come into play. For example: how many times the inlined function is used, spatial access issues (the pattern in which the code is accessed), as well as whether the compiler even pays attention to the inline request (which it is free to ignore).

As noted in the lesson text, inline is best used for functions that are no more than a couple of lines.

Abhishek
January 10, 2008 at 3:26 am · Reply

Ok