C questions

Difference between static inline function and a macro?

Static refers to the scope. In C it means that the function/variable can be only used within the same translation unit.

If the storage class is extern, the identifier has external linkage and the inline definition also provides the external definition.

If the storage class is static, the identifier has internal linkage and the inline definition is invisible in other translation units.

If the storage class is unspecified, the inline definition is only visible in the current translation unit, but the identifier still has external linkage and an external definition must be provided in a different translation unit. The compiler is free to use either the inline or the external definition if the function is called within the current translation unit.

The inline is just a compiler hint and the compiler is completely free to ignore it.

So, do you want the compiler to be aware of certain logical boundaries so it can produce better physical code, or do you want force decisions on the compiler by flattening it out manually or by using macros. The industry leans towards the former.

What is the difference between an inline function and a macro?

Inline functions are parsed by the compiler, whereas macros are expanded by the C++ preprocessor.

 An inline function typically saves the overhead of:

* Function calls (including parameter passing and placing the object's address on the stack)
* Preservation of caller's stack frame
* New stack-frame setup
* Return-value communication
* Old stack-frame restore
* Return

The fact that macros use text replacement creates the potential for bugs. Suppose we have the following code:

|  |
| --- |
| #define DOUBLE(X) X\*X  int y = 3;  int j = DOUBLE(++y); |

If you’re expecting that j will be assigned a value of 4 squared (16), then you would be wrong. Because of the text replacement, what actually happens is that the DOUBLE(++y) expands to ++y \* ++y, which equals 4\*5, giving us 20. This problem would not occur if DOUBLE were implemented as an inline function. Inline functions only evaluate their arguments once, so any side effects of evaluation happen only once.

Another problem with macros occurs with binding. Suppose we have a macro with two statements, and then we try to use that macro with an if statement. If we decide not to use the curly brackets with our if statement, then we will have something that looks like this:

|  |
| --- |
| #define ADD\_TWO(x,y) x += 2; y +=2  bool flag = true;  int j = 5, k = 7;  if(flag)  ADD\_TWO(j,k); |

Then you’re probably thinking that the macro will expand to this:

|  |
| --- |
| if(flag)  {  j +=2;  k +=2;  } |

But what actually happens is that the if statement binds to the first expression in the macro. So this is what it really expands to:

|  |
| --- |
| if(flag)  {  j +=2;  }    k +=2; |

If we had used an inline function instead of a macro, the problem shown above would not have occurred. This is because an inline function is treated as a single statement, so the entire function would be bound to the if statement.

Debugging macros is also difficult. This is because the preprocessor does the textual replacement for macros, but that textual replacement is not visible in the source code itself. Because of all this, it’s generally considered a good idea to use inline functions over macros.

Inline function follows strict parameter type checking while macros do not.