

- To inline or not to inline.
- Member versus free functions.

- Definable using the `inline` keyword or defining them inside the class.
- Means that **inline substitution** of the function body at the point of call should be preferred to the usual function call mechanism.
- Supposedly more efficient, but there are problems:
 - **code bloat**. usually, a larger executable means slower execution time because the system can cache a smaller portion of the program.
 - **dynamic linking**. it's nearly impossible to maintain binary compatibility between different versions of a binary file if an inline function's body has changed.
 - **premature optimization**. to assess whether inlining a given function improves performance, one must use a profiler and isolate the tested code from various confounding variables such as debug vs. release versions, current system load and so on. Your compiler is usually better than you at optimizing things.
- Inline functions may boost performance (in some cases dramatically), **but** in most cases they are redundant or even harmful.

MEMBER VERSUS FREE FUNCTIONS

- Consider `find_first_of`, currently a member of class `string`.

```
s.find_first_of(0x0d); // member function
find_first_of(s, 0x0d); // free function
```

Which one is better?

MEMBER VERSUS FREE FUNCTIONS

- Consider `find_first_of`, currently a member of class `string`.

```
s.find_first_of(0x0d); // member function
find_first_of(s, 0x0d); // free function
```

Which one is better?

- Both are just as readable, but the free variant can also be applied on vectors, lists, etc.
- A free function would not have access to the innards of the object so there is no danger to invalidate its state.
 - * Provided that the rest of the public interface is solid, `find_first_of` taken out can never kill your string
- Prevent the **fat interface** syndrome: a type should only have those operations that are inherently meaningful for the said type.
 - * `find_first_of` is not an essential operation for string types, it is more of a utility

RELATED POINTS

- `string` is considered a typical example of fat interface.
- Better argument can be made for `to_lower` as member function.
 - but who is to say that your string must be represented by the `string` class?
 - `vector<char>`, char arrays, MFC `CString` and other custom classes are also widely used.
 - having an iterator and an algorithm is far more flexible
- Consider passing a container to the algorithms instead of the first and last iterators.
 - you could call the iterator version from the algorithm
 - but you would then be making the assumption that all containers have begin and end member functions defined.
 - you thus rule out using the algorithms on C arrays.

CONCLUSION

- Say no to bloated interfaces.
 - Provide **type implementations** in your **classes**, with minimal interfaces.
 - The place of the **rest of your algorithm** should be in **(generic) functions**.
- Algorithm:

```
if (f needs to be virtual)
    make f a member function of C;
else if (f is operator>> or operator<<) {
    make f a non-member function;
    if (f needs access to non-public members of C)
        make f a friend of C;
}
else if (f needs type conversions on its left-most argument) {
    make f a non-member function;
    if (f needs access to non-public members of C)
        make f a friend of C;
}
else if (f can be implemented via C's public interface)
    make f a non-member function;
else
    make f a member function of C;
```

OOP: NOT A PANACEUM

I find OOP technically unsound. It attempts to decompose the world in terms of interfaces that vary on a single type. To deal with the real problems you need multisorted algebras—families of interfaces that span multiple types. I find OOP philosophically unsound. It claims that everything is an object. Even if it is true it is not very interesting—saying that everything is an object is saying nothing at all. I find OOP methodologically wrong. It starts with classes. It is as if mathematicians would start with axioms. You do not start with axioms—you start with proofs. Only when you have found a bunch of related proofs, can you come up with axioms. You end with axioms. The same thing is true in programming: you have to start with interesting algorithms. Only when you understand them well, can you come up with an interface that will let them work.