explicit-instantiation declaration in the `c.h` file will inflate the size of the `c.o` file with no possibility of reducing code bloat in client code:

```
// c.h:
#ifndef INCLUDED_C                          // internal include guard
#define INCLUDED_C

template <typename T> void f(T v) {/*...*/}  // general template definition

extern template void f<int>(int v);          // OK, matched in c.cpp
extern template void f<char>(char c);        // Error, unmatched in .cpp file

#endif


// c.cpp:
#include <c.h>                                // incorporate own header first

template void f<int>(int v);                  // OK, matched in c.h
template void f<double>(double v);            // Bug, unmatched in c.h file


// client.cpp:
#include <c.h>

void client()
{
    int    i = 1;
    char   c = 'a';
    double d = 2.0;

    f(i);  // OK, matching explicit-instantiation directives
    f(c);  // Link-Time Error, no matching explicit-instantiation definition
    f(d);  // Bug, size increased due to no matching explicit-instantiation
           // declaration.
}
```

In the example above, `f(i)` works as expected, with the linker finding the definition of `f<int>` in `c.o`; `f(c)` fails to link because no definition of `f<char>` is ~~guaranteed to be found~~ anywhere; and `f(d)` accidentally works by silently generating a *redundant* local copy of `f<double>` in `client.o`, while another, identical definition is generated explicitly in `c.o`. These extra instantiations do not result in multiply-defined symbols because they still reside in their own sections and are marked as *weak* symbols. Importantly, note that **`extern template`** has *absolutely no effect* on overload resolution ~~because~~ the call to `f(c)` ~~did~~ *not* resolve to `f<int>`.