1. Instead of using static data members, why not use global variables?

Global data members’ scope starts at the point of definition to end of file. They have external linkage, which indicates that in other source files that the same name refers to the same location in memory, whereas static members have file scope, or internal linkage. These do not conflict with other variables in other source files that share a name. If I named some data member **static int S** and another **int G** as a global variable, I would not be able to use the name S in other translation units (“source files”) to refer to the same object. Since G would have external linkage, I would be able to use this global variable to refer to this object if it’s declared as extern in another source file. S would only be available in the file (translation unit) in which it is declared.

1. Describe how it is often possible to avoid RTTI by better use of virtual functions.

One can usually take the if conditions presented in one typeid statement using RTTI and reformat them as a virtual function of an appropriate class. If there is a series of if else statements that use typeid, then you should probably stop and think whether virtual functions or dynamic\_cast may be more appropriate. If further classes are derived with such chained statements still standing, then you could find yourself in a situation where code breaks as not all possible classes in a hierarchy are accounted for. RTTI also only works for classes that have virtual functions (as these are the only hierarchies for which you could be assigning addresses of derived objects to base-class pointers) so in essence all of its tools trace back to virtual functions too. I tend to say I’m “getting lost in the sauce” when I found myself losing track of which reference values or pointers are being manipulated in a block of code that relied heavily on typeid operators or on hopes that no null pointer 0 is returned when dynamic\_casts are used, which could later break code itself.

1. Discuss the circumstances under which RTTI should be employed.

You could use RRTI when calling a specialized member function of some derived class that may not be accessible to the base class; that is, the dynamic cast operator would downcast from the base pointer to a derived pointer. We may not know at compile time whether this cast will be valid every time it is done. Another case could arise when overriding a non-virtual base class function; the dynamic cast operator would be able to cast the base pointer to derived pointer to then call a derived class’ method. RTTI comes at the cost of runtime checks but if performance loss is accounted for, it could be used more than once (though probably indicates poor design if frequently called upon).