1. Why don't conversion operators such as operator float() have return type specified?

By definition, operator float () returns a float – and invoked whenever some object needs to be implicitly converted to type float. It already has a return type defined; the return value of operator T () where T is a type is always T in C++.

We wouldn’t be able to use standard function prototype syntax such as float foo () because multiple functions with the same name which differ only by return type conflict (as float foo() conflicts with a general type T foo()). If you were to persist and use such syntax, then you can only define one conversion operator overload, which may lead to problems down the line.

1. We have talked about performing conversions in situations where you might not control the source code. In practice, do you think this is really a problem?

As previously mentioned, if the code for the destination class is not available then a conversion function has to be defined in the class of the source object. If the source class code is unavailable, then you must define a constructor in the class of the destination object. The latter leads to problems if we aren’t given enough info/accessor functions to get all the parameters from the source object that our destination class needs.

This could be an issue if you are only provided the class header details and sparse documentation concerning the implementations of a source object from a vendor, due to proprietary software. Judging from my limited experience with the professional software engineering world, I would think that vendors would not be eager to give the full implementations and “under the hood” picture of a class barring and any dependencies or add-on features that could be included at cost to better interface with the source object.

1. If you really wanted to prevent client software from performing certain conversions, prevent client software from performing certain conversions can you prevent them from happening?

You could use “explicit” to block conversions to an undesirable type if that constructor is selected, but implicit conversions in arguments don't matter. Using “delete” could also be used to block implicit conversion to an undesirable type.

A good visual is shown here: <https://stackoverflow.com/questions/36722011/prevent-undesired-conversion-in-constructor>

1. Why might the availability of automatic conversions be a problem?

Automatic conversion could lead to unintentional losses of precision depending on the direction of the conversion when dealing with certain data types:

long l = 1; // int value 1 converted to type long

double d = 1; // int value 1 converted to type double

int i = 1.23; // double value 1.23 converted to type int 3

Trying to use the “lost” fractional components or then go back to previous types that had fewer bytes in storage could lead to unintended computation down the line. Also, due to arithmetic with different types, implicit conversion of one type to another may be done without the user explicitly assigning a different allocation, such as when adding an int to a double and having the int be automatically promoted to a double for the arithmetic (perhaps using more memory than budgeted for for “larger” types).

You would not be able to not automatically convert types that are not compatible unless using a type cast. Whether this actually would make sense given the context is left up to the user to account for.

1. Why is it usually better to pass objects by reference than by value?

Passing by value forces a temporary object to be created and then assigned all the data of the source object of the same type. This is more expensive than just using the reference to that original object. Generally speaking, if a function intends to change the argument, take it by non-const reference. If the function doesn't modify the argument and the argument is of primitive type, take it by value, otherwise take it by const reference (unless if the function would then need to make a copy of the const reference).