Section 1 More Name Decoration

Study Questions

* Define the term function name decoration

The idea of an overloaded function is that you use the same function name, but different argument lists. Thus, for overloading to work the compiler must decorate the function name with the names of the argument types.

* Describe why return values cannot be overloaded in C++.

However, in C you’ve always been able to call a function and ignore the return value (that is, you can call the function for its *side effects).* How can the compiler distinguish which call is meant in this case? Possibly worse is the difficulty the reader has in knowing which function call is meant. Overloading solely on return value is a bit too subtle, and thus isn’t allowed in C++.

* Define the term type-safe linkage.

 if for some reason you still manage to misdeclare a function, either by declaring by hand or including the wrong header file (perhaps one that is out of date), the name decoration provides a safety net that is often referred to as *type-safe linkage.*

Even though you can see that the function is actually **f(int)**, the compiler doesn’t know this because it was told - through an explicit declaration - that the function is **f(char).**Thus, the compilation is successful. In C, the linker would also be successful, but *not* in C++. Because the compiler decorates the names, the definition becomes something like **f\_int**, whereas the use of the function is **f\_char.**

Although the problem doesn’t occur all that often, when it does it can be incredibly difficult to find, especially in a large project. This is one of the cases where you can easily find a difficult error in a C program simply by running it through the C++ compiler.

Section 2 Overloading Example

When you have completed this section, you should be able to observe the use of overloading in the example program Stash.

Yu can use two different approaches to enter the same ideation part, but different applications for it