What with function overloading, function templates, and function template overloading, C++ needs, and has, a well-defined strategy for deciding which function definition to use for a function call, particularly when there are multiple arguments. The process is called overload resolution. Detailing the complete strategy would take a small chapter, so let's take an overview of how the process works:

- Phase 1: Assemble a list of candidate functions. These are functions and template functions with the same name as the called function.

- Phase 2: From the candidate functions, assemble a list of viable functions. These are functions with the correct number of arguments and for which there is an implicit conversion sequence, which includes the case of an exact match for each type of actual argument to the type of the corresponding formal argument. For example, a function call with a type `float` argument could have that value converted to a `double` to match a type `double` formal parameter, and a template could generate an instantiation for `float`.

- Phase 3: Determine if there is a best viable function. If so, use that function. Otherwise, the function call is an error.

Consider a case with just one function argument, for example, the following call:

\[ \text{may('B');} \quad // \text{actual argument is type char} \]

First, the compiler rounds up the suspects, which are functions and function templates having the name `may()`. Then, it finds those that can be called with one argument. For example, the following will pass muster because they have the same name:

\[
\begin{align*}
&\text{void may(int);} \quad // \#1 \\
&\text{float may(float, float = 3);} \quad // \#2 \\
&\text{void may(char);} \quad // \#3 \\
&\text{char * may(const char *);} \quad // \#4 \\
&\text{char may(const char &);} \quad // \#5 \\
&\text{template<class T> void may(const T &);} \quad // \#6 \\
&\text{template<class T> void may(T *);} \quad // \#7
\end{align*}
\]

Note that just the signatures and not the return types are considered. Two of these candidates (#4 and #7), however, are not viable because an integral type cannot be converted implicitly (i.e., without an explicit type cast) to a pointer type. The remaining