Another case where one exact match is better than another is when one function is a non-template function and the other isn't. In that case, the non-template is considered better than a template, including explicit specializations.

If you wind up with two exact matches that both happen to be template functions, the template function that is the more specialized, if either, is the better function. That means, for example, that an explicit specialization is chosen over one generated implicitly from the template pattern:

```c
struct blot {int a; char b[10];};
template <class Type> void recycle (Type t); // template
template <> void recycle<blot> (blot & t);   // specialization for blot
...
blot ink = {25, "spots"};
...
recycle(ink);  // use specialization
```

The term "most specialized" doesn't necessarily imply an explicit specialization; more generally, it indicates that fewer conversions take place when the compiler deduces what type to use. For example, consider the following two templates:

```c
template <class Type> void recycle (Type t);    #1
template <class Type> void recycle (Type * t);..#2
```

Suppose the program that contains these templates also contains the following code:

```c
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(&ink);  // address of a structure
```

The `recycle(&ink)` call matches template #1 with Type interpreted as `blot *`. The `recycle(&ink)` function call also matches template #2, this time with Type being `ink`. This combination sends two implicit instantiations, `recycle<blot *>(blot *)` and `recycle<blot>(blot *)`, to the viable function pool.

Of these two template functions, `recycle<blot *>(blot *)` is considered the more specialized because it underwent fewer conversions in being generated. That is, template