cout << cube(x);

The x argument matches both the double x prototype and the double &x prototype. Thus, the compiler has no way of knowing which function to use. Therefore, to avoid such confusion, when it checks function signatures, the compiler considers a reference to a type and the type itself to be the same signature.

The function matching process does discriminate between const and non-const variables. Consider the following prototypes:

```c
void dribble(char * bits);       // overloaded
void dribble(const char *cbits); // overloaded
void dabble(char * bits);         // not overloaded
void drivel(const char * bits);   // not overloaded
```

Here's what various function calls would match:

```c
const char p1[20] = "How's the weather?";
char p2[20] = "How's business?";
dribble(p1);       // dribble(const char *);
dribble(p2);       // dribble(char *);
dabble(p1);       // no match
dabble(p2);       // dabble(char *);
drivell(p1);      // drivel(const char *);
drivell(p2);      // drivel(const char *);
```

The dribble() function has two prototypes, one for const pointers and one for regular pointers, and the compiler selects one or the other depending on whether or not the actual argument is const. The dabble() function only matches a call with a non-const argument, but the drivell() function matches calls with either const or non-const arguments. The reason for this difference in behavior between drivell() and dabble() is that it's valid to assign a non-const value to a const variable, but not vice versa.

Keep in mind that it's the signature, not the function type, that enables function overloading. For example, the following two declarations are incompatible:

```c
long gronk(int n, float m);   // same signatures,
double gronk(int n, float m); // hence not allowed
```