The Meaning of "f(x)" in C++

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/638-6614

Function Calls and Implicit Type Conversions

Consider:

```
void f(double d);

int x;

...

f(x);                    // call f with an int
```

Should this compile?

- x is of the wrong type.

C says yes. So does C++.

- Note: this is *an attempt to read minds*.

Function Calls and Overloading

Consider:

```
void f(int);
void f(double);
```

Should this compile?

- f is overloaded

C++ says yes.

Overloading Meets Type Conversions

Now consider an abstract view of a set of overloaded functions and a potential call:

```
void f(SomeParamType1);
void f(SomeParamType2);
...
void f(SomeParamTypeN);

SomeType x;
f(x);                          // A call to f, but which one?
```

C++ specifies five levels of parameter matching that can be applied:

1. Exact match (includes "trivial conversions")
2. Match with promotions (value-preserving)
3. Match with standard conversions (not always value-preserving, includes inheritance-based conversions)
4. Match with user-defined conversions
5. Match with ellipsis

Resolving Function Calls

These rules largely determine which, if any, function should be called. Example:

```
void f(int);
void f(int*);
void f(...)

f(10);                  // calls f(int) — exact match
f(0);                   // calls f(int) — exact match

string *ps = new string;

f(ps);                  // calls f(...) — match with ellipsis
```

Functions taking multiple parameters do the same thing, only more so.

- For a call to compile, the called function must:
  - Be at least as good a match on each parameter as all the other candidate functions and
  - Be a strictly better match on at least one parameter.

Note: this is still *an attempt to read minds*.

Implicit Template Type Deduction

Consider:

```
template<typename T>
void f(T);

int x;

f(x);                   // Deduce that this is a call to f<int>
```

Note that no type conversion is ever necessary.

- T can always be the passed type.

Implicit Template Type Deduction

It gets more interesting with *one type parameter* but *multiple function parameters*:

```
template<typename T>
void f(const T& x, const T& y);
```

Should mixed-type calls compile?

```
int i;
const int ci = 5;

f(i, ci);                    // Valid? If so,what is T?


double d;

f(i, d);                     // Valid? If so,what is T?
```


Implicit Template Type Deduction

And of course there is the inheritance issue:

```
class Base { ... };
class Derived:
   public Base { ... };

Derived d;
Base& rb = d;

f(rb, d);                    // Valid? If so,what is T?
```

Type Conversions and Implicit Template Type Deduction

C++ allows some type conversions during implicit type deduction:

- The first and third examples are legal. The second is not.

The allowed conversions are more constrained than for function calls:

- Exact match (with some "trivial conversions")
- Match with inheritance-based conversions

What's missing?

- Promotions
- Standard conversions other than inheritance-based ones
- User-defined conversions

Note: again, this is *an attempt to read minds*.

The Crux of the Issue

Consider:

```
f(x);                    // What is this?
```

Is this a function call?

- If so, conversion rules for function calls apply.

Is it a request to instantiate and call a template function?

- If so, conversion rules for template instantiation apply.

The Rubber Hits the Road

The problem is not purely theoretical:

```
void f(vector<int>::const_iterator it1, vector<int>::const_iterator it2);

vector<int> v;
...
vector<int>::iterator begin = v.begin();
vector<int>::const_iterator end = v.end();

f(begin, end);          // fine, this is a function call, so the user-defined
                        // iterator ⇒ const_iterator conversion applies


template<typename It> void g(It it1, It it2);

g(begin, end);          // error, this is a template instantiation, so
                        // no user-defined conversions apply;
                        // no type for It can be deduced.
```


Specializing Templates

Aber warten Sie mal, wir gehen noch weiter.

It often makes sense to specialize templates for one or more types:

```
template<typename T>
void f(T);                  // General template

template<typename T>
void f(T*);                 // General Template For Pointers

template<>
void f<char*>(char *p);     // Template specialization for char*
                            // pointers. This is not a template.
```

This turns out to be useful. Really :-)

Specializing Templates

Consider:

```
template<typename T>
void f(T);                      // (1) General Template

template<typename T>
void f(T*);                     // (2) General Template for Pointers

template<>
void f<char*>(char *p);         // (3) Specialization of (1)
                                // for char* Pointers

char *p;
...
f(p);                           // Which f is instantiated/called?
```


Specializing Templates

Critical observations:

- Only *functions* can be called.
- *Function templates* are not functions. They *generate* functions.
- Before the compiler generates a function, it must choose the *template* to instantiate.

There are only two templates to choose from:

```
template<typename T>
void f(T);                      // (1) General Template

template<typename T>
void f(T*);                     // (2) General Template for Pointers
```

Here is the call again:

```
char *p;
...
f(p);                           // Which f is instantiated/called?
```

Which template is a better match for a pointer type?

Specializing Templates

Clearly, the template for pointers is a better match. So:

```
template<typename T>
void f(T);                      // (1) General Template

template<typename T>
void f(T*);                     // (2) General Template for Pointers

template<>
void f<char*>(char *p);         // (3) Specialization of (1)
                                //     for char* Pointers
char *p;
...
f(p);                           // Calls (2), not (3)
```

The specialization would be considered only if (1) were the selected template!

The results would change if (3) were declared this way:

```
template<>
void f<char>(char *p);          // Now this specializes (2), not (1)!
```


Resolving Function Calls

In essence, there are three sets of interacting rules:

- Overloading resolution
- Template argument deduction
- Function template partial ordering

All may apply to what looks like a simple function call:

```
f(x);                   // all of the above may be involved
```

Implications for C++ Programmers

- You must know whether you are using a template name when making a function call.

```
f(x);          // what happens here depends on whether f is
               // a function name, a template name, or both
```

- You must document whether functionality you provide comes from functions or function templates.
- Be careful not to confuse template argument deduction with overloading resolution.
  - ➠ This applies also to non-type template arguments. The conversion rules for those also differ from those for overloading resolution.

Implications for Language Designers

- If X is a good idea and Y is a good idea, X+Y is not necessarily a good idea.
- The road to language Hell is paved with good intentions.
- It's hard to read minds.