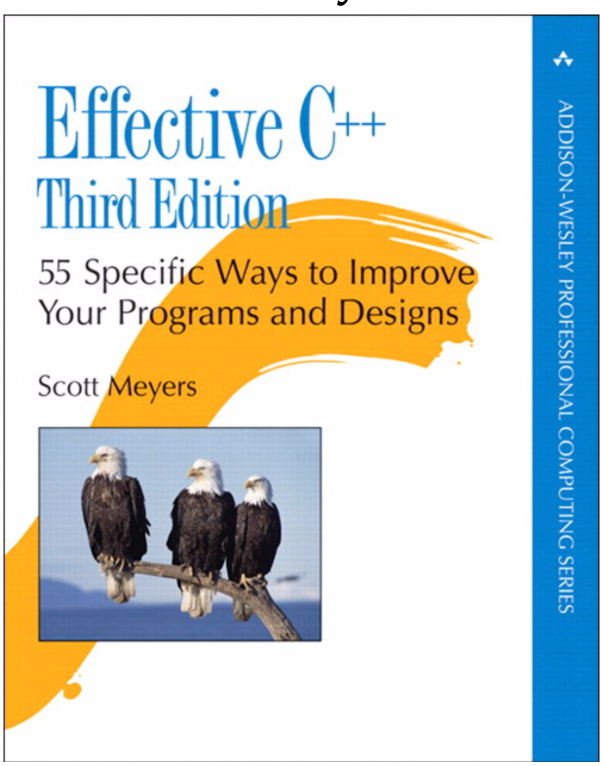
The Meaning of "f(x)" in C++

Scott Meyers, Ph.D.
Software Development Consultant

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

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

Buy This Book!



The Mark of a True Professional!

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.

Dimensional Analysis in C++

Scott Meyers, Ph.D.
Software Development Consultant

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

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

Enforcing Dimensional Unit Correctness

Scientific and engineering calculations are dependent on correct use of units in calculations:

- It makes no sense to assign a time value to a distance variable
- It makes no sense to compare a mass variable with a charge variable

But most software ignores such units:

```
double t;                 // time - in seconds
double a;                 // acceleration - in meters/sec2
double d;                 // distance - in meters
...
cout << d/(t*t) - a;      // okay, subtracts meters/sec2
cout << d/t - a;          // should be an error, as it
                          // subtracts meters/sec and
                          // meters/sec2
```


Enforcing Dimensional Unit Correctness

Typedefs just disguise the problem:

```
typedef double Acceleration;
typedef double Time;
typedef double Distance;

Time t;
Acceleration a;
Distance d;

...

cout << d/t - a;          // still compiles, but is still wrong
```

We want a way to use the C++ type system to:

- Make unit compatibility errors impossible:
  - They'll be detected during compilation
- Do so with minimal runtime performance impact:
  - Minimal memory overhead, minimal runtime overhead
  - As much as possible should be done during compilation

Enforcing Dimensional Unit Correctness

Observations:

- The number of needed types is, in principle, unlimited:
  - ➠ Time * Time = $Time^2$
  - ➠ Time/Distance = Time/Distance
  - ➠ Distance/$Time^2$ = Distance/$Time^2$
- This suggests we should have templates generate the types automatically.
- Types change only when a unit type's *exponent* changes:
  - ➠ Unitless numbers (i.e. constants) have unit exponents of 0
  - ➠ In Time * Time, the Time exponent goes from 1 to 2
  - ➠ In Acceleration/Time, the Time exponent goes from -2 to -3
- This suggests we need a template to generate types based on unit exponents

Enforcing Dimensional Unit Correctness

```
template<int m,                          // exponent for mass
         int d,                          // exponent for distance
         int t>                          // exponent for time
class Units {
public:
  explicit Units(double initVal = 0): val(initVal) {}

  double value() const { return val; }
  double& value() { return val; }
  ...

private:
  double val;
};
```

Now we can say:

```
Units<1, 0, 0> m;                        // m is of type mass
Units<0, 1, 0> d;                        // d is of type distance
Units<0, 0, 1> t;                        // t is of type time

m = t;                                   // error! type mismatch
```

Enforcing Dimensional Unit Correctness

Typedefs for commonly-used units make things clearer:

```
typedef Units<1, 0, 0> Mass;
typedef Units<0, 1, 0> Distance;
typedef Units<0, 0, 1> Time;

Mass m;
Distance d;
Time t;
```


Enforcing Dimensional Unit Correctness

Arithmetic operations on these kinds of types are important, so we can augment Units as follows:

```
template<int m, int d, int t>
class Units {
public:
  ...                                   // as before

  Units<m, d, t>& operator+=(const Units<m, d, t>& rhs)
  {
    val += rhs.val;
    return *this;
  }

  Units<m, d, t>& operator*=(double rhs)
  {
    val *= rhs;
    return *this;
  }

  ...
};
```

Operators for subtraction and division are analogous.

Enforcing Dimensional Unit Correctness

Non-assignment operators are best implemented as non-members:

```
template<int m, int d, int t>
const Units<m, d, t> operator+(const Units<m, d, t>& lhs,
                               const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(lhs);
  return result += rhs;
}
template<int m, int d, int t>
const Units<m, d, t> operator*(double lhs,
                               const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(rhs);
  return result *= lhs;
}
template<int m, int d, int t>
const Units<m, d, t> operator*(const Units<m, d, t>& lhs,
                               double rhs)
{
  Units<m, d, t> result(lhs);
  return result *= rhs;
}
```


Enforcing Dimensional Unit Correctness

If we adopt the SI units as our standard, we can provide the following constants:

```
const Mass kilogram(1);         // each of these constants sets its
const Distance meter(1);        // internal val field to 1.0
const Time second(1);
```

Now we can start defining more interesting objects:

```
Distance myBatikHeight(0.5 * meter);
Distance myBatikWidth(1 * meter);

Mass willametteMeteoritesWeight(13636 * kilogram);

Time halfAMinute(30 * second);
```

Enforcing Dimensional Unit Correctness

We can also define other units in terms of our standard:

```
const Mass pound(kilogram/2.2);
const Mass ton(907.18 * kilogram);
const Time minute(60 * second);
const Time hour(60 * minute);
const Time day(24 * hour);
const Distance inch(.0254 * meter);
```


Enforcing Dimensional Unit Correctness

The real fun comes when multiplying/dividing Units:

```
template< int m1, int d1, int t1,
          int m2, int d2, int t2>
const Units<m1+m2, d1+d2, t1+t2>
operator*( const Units<m1, d1, t1>& lhs,
           const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1+m2, d1+d2, t1+t2> ResultType;

  return ResultType(lhs.value() * rhs.value());
}

template< int m1, int d1, int t1,
          int m2, int d2, int t2>
const Units<m1-m2, d1-d2, t1-t2>
operator/( const Units<m1, d1, t1>& lhs,
           const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1-m2, d1-d2, t1-t2> ResultType;

  return ResultType(lhs.value() / rhs.value());
}
```

Enforcing Dimensional Unit Correctness

Real implementations typically use more template arguments for Units:

- One specifies the precision of the value (typically float or double)
- The others are for the exponents of the seven SI units:
  - Mass
  - Length
  - Time
  - Charge
  - Temperature
  - Intensity
  - Angle

Enforcing Dimensional Unit Correctness

```
template<class T, int m, int d, int t, int q, int k, int i, int a>
class Units {
public:
  explicit Units(T initVal = 0) : val(initVal) {}
  T& value() { return val; }
  const T& value() const { return val; }
  ...
private:
  T val;
};

template<class T, int m1, int d1, int t1, int q1, int k1, int i1, int a1,
                  int m2, int d2, int t2, int q2, int k2, int i2, int a2>
Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
operator*(const Units<T, m1, d1, t1, q1, k1, i1, a1>& lhs,
          const Units<T, m2, d2, t2, q2, k2, i2, a2>& rhs)
{
  typedef Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
          ResultType;

  return ResultType(lhs.value() * rhs.value());
}
```


Observations

Dimensionless quantities (i.e., objects of type Units<T, 0,0,0,0,0,0,0>) should be type-compatible with unitless types (e.g., int, double, etc.).

- Partial template specialization can help:

```
template<typename T>
class Units<T, 0, 0, 0, 0, 0, 0, 0> {
public:
  ...
  Units(T initVal = 0): val(initVal) {}        // allow implicit conversion
  operator T() const { return val; }           // to/from values of type T

  Units& operator=(T newVal)                   // allow assignments from
  { val = newVal; return *this; }              // values of type T
  ...

private:
  T val;
};
```

If partial template specialization is unavailable, you can totally specialize for e.g., T = double and/or T = float.

Observations

Some compilers refuse to place objects in registers:

- A Units<double, ...> may thus be treated less efficiently than a raw double
- If efficiency is a problem, you can revert to type-unsafe typedefs:

```
typedef double Acceleration;
typedef double Time;
typedef double Distance;
```

➠ This is okay as long as the code has already been shown to compile using Units

Industrial-Strength Dimensional Analysis

A state-of-the-art implementation of the Units approach is more efficient, powerful, and sophisticated:

- It allows fractional exponents (e.g., distance$^{1/2}$)
- It supports multiple unit systems (beyond just SI)
- It uses template metaprogramming to shift some computation from runtime to compiletime.
  - E.g., to compute GCDs when reducing fractional exponents.

Industrial-Strength Dimensional Analysis

It can determine whether this "simple" formula,

$$\frac{1}{X_0} = 4\,\alpha\,r_e^2\,\frac{N_A}{A}\,\left\{Z^2\,\left[L_{rad} - f(Z)\right] + Z\,L'_{rad}\right\}$$

is correctly modeled by this C++:

```
Energy<> finalEnergy(Element<> const & material, Density<> const dens,
                     Length<> const thick, Energy<> const initEnergy) {
  AtomicWeight<> const A = material->atomicWeight;
  AtomicNumber<> const Z = material->atomicNumber;

  Number<> const L_rad = log( 184.15 / root<3>( Z ) );
  Number<> const Lp_rad = log( 1194. / root<3>(Z*Z) );

  Length<> const X_0 = 4.0 * alpha * r_e * r_e * N_A / A *
                       ( Z * Z * L_rad + Z * Lp_rad );

  return initEnergy / exp( thick / X_0 );
}
```

(It's not. There are three dimensional type errors.)

Conclusions

- Templates are useful for a lot more than just containers
- Templates make it possible to generate and check an unknowable number of types during compilation
- Templates can add type safety to code with little or no runtime penalty

Further Reading

- John J. Barton and Lee R. Nackman, "Dimensional analysis," *C++ Report*, January 1995. Based on section 16.5 of their *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, 1994, ISBN 0-201-53393-6.
  - ➠ Now primarily of historical interest.
- Walter E. Brown, "Introduction to the SI Library of Unit-Based Computation," International Conference on Computing in High Energy Physics (CHEP '98), August 1998. Available at http://fnalpubs.fnal.gov/archive/1998/conf/Conf-98-328.pdf.
  - ➠ A user's view of SIUNITS. Describes how five different models of the universe are supported.
- Walter E. Brown, "Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation," Second Workshop on C++ Template Programming, October 2001. Available at http://www.oonumerics.org/tmpw01/brown.pdf.
  - ➠ Another description of SIUNITS, this time focusing more on implementation strategies.

Further Reading

- Michael Kenniston, "Dimension Checking of Physical Quantities," *C/C++ Users Journal*, November 2002.
  - ➠ A description of a slightly different approach, one focused on working with less conformant compilers (e.g., Visual C++ 6).

And of course:

- Scott Meyers, *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 2005, ISBN 0-321-33487-6.