An attempt at defining ambiguity resolution for C++.

First the general principles:

-Typing in C++ is bottom-up: the type of a subexpression is independent of its context. Since the type of variables and litterals are explicit, specifying C++ type resolution reducess to defining how the type of a function or operator call is resolved, given the types of the arguments or operands.

-Mandatory conversions (type adjustments):
  -Following ANSI C, expressions yielding a function or array type are ALWAYS converted to pointer type except in the syntactic context of the & or sizeof operators. Also, for the purposes of type-checking, argument type that are function or array types are adjusted to be pointer types. This is done prior to the value to reference adjustments defined below. (References to arrays or functions must be barred to make this consistent.)
  -In C++ value (non-reference) formal arguments of type T can match any T& actual argument, regardless of the qualifiers applied to T in either case. Conversely, C++ allows one to bind a const T& or const volatile T& to the result of a function returning a T value regardless of any qualification of that value. To express this, we will assume that
    -For the purposes of matching a prototype to a function call, any return type (qualified) T, where T is a non-reference, is adjusted to const T&.
    -For the purposes of matching a prototype to a function call, any formal argument type (qualified) T, where T is a non-reference, is adjusted to const volatile T&.
    -These adjustments are NOT done inside function types, which retain their original form.
    -For the purposes of type checking, any variable or data member (qualified) T, where T is a non-reference, is adjusted (qualified) T&. This also applies to formal arguments when type-checking the corresponding function body.
    -For the purposes of type checking, any constant of type T is considered to have type const T&.
  To avoid cluttering the rest of this description, although we will assume that these adjustments are always done, we will continue to describe functions (including conversions) in their unadjusted form (so when we talk about the int(float) conversion, we actually mean the const int& (const volatile float&) conversion).
-Now on to overloading resolution. To resolve a function or operator call we consider all the functions or operators that match the name of that function or operator, within the following domains:
  -In a global function call, we consider only global functions, except if the function name is hidden by a class member (in a class context). This also applies for explicit operator calls (e.g., operator+(x,y)), with the same exception for class contexts.
  -In a '.' member call (object.member(args)), the type of object must be a class X, possibly qualified, and posssibly referenced. Only those members found in class X by the class lookup rules are considered. The call is ambiguous if the lookup is ambiguous. This rule also applies if the member is an operator (as in x.operator+(y)), and for direct members in a class context (e.g., f() in void X::f() { f(); }).
  -In an infix call to new we consider either the the class operator new if any is defined, or the global one by default. On the other hand, all the operator delete are considered in an infix delete call.

-In a '->' member call (p->member(args)), the type of 'p' must be either a (qualified) pointer to (qualified) class X, or a (a reference to) a (qualified) class Y with operator '->'. In the latter case, applying repeatedly operator -> must yield a (qualified) pointer to some class X. In both cases, only the members obtained by class lookup in X are considered.
-In an infix operator call (e.g, x+y, x-=y, x[y]), if the leftmost argument is of type (reference to) (qualified) class X, only global operator op and operator op obtained by lookup in class X are considered. This does not apply to explicit call (operator+(x,y)) where only the global operator is considered. I am not sure of what should become of unary operator& and operator, (are the predefined meanings erased?).
-Functions or members with ellipses are considered only if there are no matchings with any other functions, and a function matching n arguments against the ellipsis is considered only if there are no matchings.

The argument lists of the call and/or members functions considered are modified as follows, for the purpose of type checking:

-A function with n default arguments is considered to be n+1 different functions.
-Class X non-static members are considered to take an X& from which the 'this' pointer is derived; the derivation of the 'this' pointer is independant of any operator& definition for class X (indeed it is the only way of taking the address of an X object whose & operator has been redefined). This extra argument will take on any member qualifiers (e.g., it will be a const X& for a const member. For the purposes of type checking, the call is assumed to provide the extra argument, as either
    -'x' in x.member(args), op x, x op, x op y, x(args), x[y], x.*y, or x->*y,
    -'*x', '*x.operator->()', '*x.operator->().operator->()', ..., whichever is appropriate, x->member(args),
    -'*this' in a direct member call.
-In calls to new or delete the size_t argument does not affect type-checking, and can be safely ignored.
-A class A::operator delete definitions behave as if it declared a global operator delete (const class A*const&); the global delete is checked as if it were delete (const void*const&).

-Given the set of considered functions and the tuple of call argument types, we can define the set of MATCHINGS for that call:
-A matching is considered function together with a tuple of admissible sequences of conversions that convert each call argument type to the function argument type. Conversions are listed below. Admissible sequences have at most one user-defined conversion, see below for a precise definition.
-If there are no matchings the call is a MISMATCH. If there is only one matching then the call is unambiguous. Otherwise we need to use the ambiguity resolution rules described below.
-To help resolve ambiguities, conversions are classified in a partial order. There are two grades of ordering, used as follows:
    -when a conversion C is locally better than a conversion C' (denoted C>C'), we will prefer to perform C rather than C' for a given argument.
    -when a conversion of C is globally better than a conversion C' (denoted C>>C'), we will prefer to perform C on one argument if this avoids doing C' on any other argument.

Of course globally better implies locally better.
-We define a preference preorder on matching as follows:
    -given two matchings M and M', we define the two distinguishing tuples D and D' as follows:
        for all i, D_i is a maximal (for <) element of

```
            {C in M_i | C not in M'_i}
        or $exact if this set is empty ($exact is the exact matching).
        For this definition, we consider that C is in M'_i when:
           -C appears as is in the M'_i sequence
           -C is a qualifier conversion and is included in another qualifier
            conversion in the M'_i sequence (e.g., char*&->char*const& is
            included in char*&->const char*const&) when volatile qualifiers
            are ignored.
           -C is a user conversion to a type T, M'_i contains a user conversion
            to a type T', and T' can be converted to T without user conversions
            or T can be converted to T' without user conversions.
     -we say we prefer M to M' if some conversion in D that is locally better
      than the corresponding conversion C in D' and C is not globally better
      than any conversion in D. Mathematically,
         exists i, D_i > D'_i and not exists j, D'_i >> D_j
 -If there is a maximum matching for the preference preorder (i.e., a matching
  that is preferred to all others and to which no other is preferred), then that
  matching is chosen. If the matching contains an ambiguous user conversion,
  then the call is also ambiguous; if the matching requires a non-existent
  copy constructor (e.g., only T::T(T&) exists), the the call is a mismatch.
  Otherwise the call is unambiguous.

-Now the list of conversions; here a naked 'better' will mean globally better.
   -The best conversion is no conversion at all, that is the exact match $exact.
   -Qualifier conversions: The most elementary conversions are adding const or
    volatile qualifiers to a referenced or pointed-to type; these can be added
    simultaneously at any level of indirection. Qualifier conversions are better
    than any of the other conversions, but exact match and volatile qualifier
    conversions are better than const qualifier conversions. The exact match is
    not better than the volatile qualifier conversions, so the volatile qualifier
    does not help to lift ambiguities.
   -Standard conversions: There are two categories of standard conversions,
    conversions between arithmetic types and conversions between pointer or
    reference types. In arithmetic conversions, integral promotions and float to
    double conversions are globally better than any other standard conversion;
    also, conversions that are arithmetic promotions (e.g., int to float or short
    to unsigned short) are globally better than those that are not (e.g., long
    double to char). Within pointer conversions, those from a class A* to a
    class B* are locally better than those from a class A* to a C* if C is a
    base class of B or void. Similarly for reference, and pointer to member
    conversions (all these cnversions preserve qualifiers). The other kinds of
    standard conversions are 0 to null pointer and pointer to function to void*.
   -Conversions to references: for now, C++ still allows to convert a const& to
    a & by introducing a temporary variable. However, this is not allowed
    for the this argument of members, except in infix operator calls. Also,
    this does not allow one to convert a volatile& to a &.
   -User conversions: Both standard conversions and conversions to references
    are better than user-defined conversions. User conversions include not only
    those conversions actually defined by the user, but all those that can be
    performed by using standard, or const, or value conversions followed by a user
    conversions. The resolution of such composite conversions is done recursively
    according to the macthing rules above. Some user conversions may be ambiguous;
    this does not affect the call matching resolution, except that the whole call
    becomes ambiguous if an ambiguous conversion is selected.
   -Ellipsis conversions: Finally, any value can be converted to match the
    ellipsis, but any conversion is better than that (this principle is already
```

embodied in the function selection rules given above).

-Admissible sequences: They are restricted by the following rules
-at most one user conversion (but it includes any sequence of standard, qualifier, or reference required to apply the user-defined conversion)
-at most one standard conversion.
-no qualifier conversion after a qualifier conversion (they can be bundled together), or after a pointer standard conversion (they cna be perfomed beforehand).

Examples:

Bottom-up typing:

```
class FLOAT {};
FLOAT f (float);
int f (int);
float x;       // could be resolved as f(int(x)), but is rejected
int y = f(x);  // because of bottom-up resolution
```

Mandatory conversions: function types

```
int d();
int (*p_d) = ********************d;
```

Adjustment within argument types:

```
typedef int A[10];
int h (A a);
int g (int (*f) (int*));
int g_f = g(f); // ok, h matches the f prototype
```

Mandatory conversions: value-to-reference adjustments

```
int f (int);
int f (int&);
int x;
int f_1 = f(1);      // ok, f(int), adjusted to f(const volatile int&)
int f_x = f(x);      // ok, f(int&)
int f_f_x = f(f(x)); // ok, f((const volatile int&)f((int&)x))
```

Global functions vs class context:

```
void f(int);
class X { void f(); };
void X::f() {
   f(); // uses X::f
   f(0); // mismatch
   extern void f (int);
   f(); // mismatch
   f(0); // calls ::f(int)
}
```

Direct member lookup:

```
struct X {
   int f ();
};
int g (X&,int);
struct Y : X {
   int f (int);
} y;
int g (Y&);
```

```
   int y_f = y.f(); // fails because class lookup does not get X::f
   int g_y = g (y); // succeeds because standard conversion is allowed

Domain for new and delete:
   struct X {
      void* operator new (size_t,int);
      void operator delete (void*);
   }
   X* p_x = new X; // fails, ::new not considered
   struct Y {
      operator X* ();
   };
   main() {
     const X* p_x;
     delete p_x; // ok for X::delete;
     const int* p_i;
     delete p_i; // calls global delete, 2.1 gets this one wrong!
     Y y;
     delete y; // ok, calls X::delete ((void*)(X*)Y);
   }

Indirect member call: using operator->
   struct X { int m(); };
   struct Y {
      X* operator->();
   } y;
   int y_m = y->m(); // calls x->m();

Operator resolution: members vs globals
   struct X {
      X (int);
      int operator+ (int);
   };
   struct Y : X {
      Y();
      int operator+ (Y);
   } y;
   int operator+ (X&,X);
   int y_plus_0 = y+0;  // resolved to operator+((X&)Y,X(0))
                        // y.X::operator+(0) not considered

Ellipses:
   int f (int);
   int f (int, int, ...)
   int f (int, ...);
   int f_0 = f(0); // ok, f(int) has no ellipsis so f(int,...) is not considered
   int f_0_0_0 = f (0,0,0); // ok, f(int,int,...) matches only one argument
                            // with the ellipsis, so f(int,...) is not considered

Argument modification: default arguments
   struct X { X(int); };
   int f (int, complex = 0); // considered as both f(int) and f(int,complex)
   int f (double);
   int f_0 = f(0);  // ok, f(0,complex(0)), user conversion in default argument
                    // does not interfere with choice of function
```

```
Argument modification: the this argument
   struct X {
      int m () const;
      int& m();
   } *x;
   const X *y;
   int& x_m = x->m(); // ok, 'this' argument selects int& m();
   int& y_m = y->m(); // warning, 'this' argument selects int m();
                      // needs reference conversion.

Example of Matchings:
   struct X {
      X (int);
   };
   int f (X&, X,   double, int);       // function f1
   int f (X&, X,   int,    double);    // function f2
   int f (X&, int, double, double);    // function f3
   int f_1e2_0_1_1 = f (1e2, 0, 1, 1); // call to match
matchings:
 -To simplify things, we'll use X, int, double for const X&, const int&, and
  const double&. Also, we'll omit the volatile qualifier conversions, since they
  don't affect the resolution here.
   f1 (double->X->X&, int->X, int->double, $exact)
   f2 (double->X->X&, int->X, $exact,      int->double)
   f2 (double->X->X&, $exact, int->double, int->double)
note that the first user conversion actually resolves to the sequence
   double -> int -> X
preference preorder:
  -Between f1 and f2 we have
     f1' = ($exact, $exact, int->double, &exact)
     f2' = ($exact, $exact, &exact, int->double)
   thus we prefer f1 over f2 because of the last argument ($exact is better
   than int->double, a standard conversion, and int->double is not better
   than any component of f1'). However, we also prefer f2 over f1 because
   of the third argument. This there is an ambiguity between f1 and f2.
  -However if we consider f1 and f3 we have
     f1' = ($exact, int->X, &exact, &exact)
     f3' = ($exact, $exact, &exact, int->double)
   Then we prefer f3 over f1 because of the second argument, but we do prefer
   f1 over f3: the last argument is the only place where f1' is locally better
   than f3', but then the int->double standard conversion is globally better
   than the int->X conversion in f1'.
  -Since the same holds between f2 and f3, f3 is the maximum for the preference
   preorder and the call is unambiguous. Note that the fact that the first
   argument requires a lot of conversions does not affect the ambiguity
   resolution, since the required conversions are the same in all cases.

The membership rules: identifying user conversions:
   struct X {
      operator int();
      operator float();
   } x;
   double d = x; // X::operator float() preferred, because the difference sets
                 // are (float->double) which is preferred to (int->double)
The reciprocal rule (T can be converted to T') is needed to handle the odd case
where the conversions return a volatile& and a &, since a volatile& cannot be
```

converted to a &.

```
Ambiguous user conversion:
   struct X {
      X (float);
      X (short);
   };
   int f (X);
   int f_0 = f(0); // selects int->X unambiguously, but int->X is ambiguous
                   // by itself
Unavailable constructor: also includes
   struct X {
      X (const X&);
   };
   int f(X);
   volatile X x;
   int f_x = f(x);


Motivations for the conversion ordering:

Volatile does not influence resolution:
   int f(int); // adjusted to f(const volatile int&)
   int f(const int&);
   int f_0 = f(0); // ambiguous, volatile conversion does not help lift ambiguity

See above for examples where const lifts ambiguity;

Integral promotions are better:
   struct A {};
   struct B : A {};
   int f (A*, char);
   int f (B*, int);
   B* b;
   int f_b_B = f(b,'B'); // use f (B*, int), simple integral promotion better
                         // than reverting to base class
But not other arithmetic conversions:
   int f(A*,double);
   int f_b_1e2 = f(b,1e2); // ambiguous, B*->A* or double->int?
Integral promotions are better than standard conversions:
   int f(B*, float);
   int f_b_C = f(b,'C'); // ok, char->int rather than char->double
I see no good reason this should not be "locally better".

Extensions are better than truncations:
   int plus (int,int);
   double plus (double, double);
   double plus_1e2_1e3 = plus (1e2,1e3);
        // ok, int->double better than double->int

Local preferences among pointer conversions emulate inheritance:
   struct C : B {};
   int f(A*);
   int f(B*);
   C* c;
   int f_c = f(c); // calls f((B*)c);
```

```
int g(void*);
int g(A*);
int g_c = g(c); // calls f((A*)c);
```

I see no specific examples for making user conversions and ellipses worse than all others, except a general fear and distrust for these facilities. (This may be justified by the fact that almost anything matches the ellipsis, and that int can be converted by constructor to almost anything that needs a size parameter.)