Combat line alone not conducive to creating consensus.
Claims of incomplete specification exaggerated

L10

Grammatical Considerations for C++
X3J16/90-0084

Mark Langley
Microsoft Corp
1 Microsoft Way
Redmond, WA 98052

uunet!microsoft!marklan

Abstract

IS does

This paper discusses syntactic and semantic ambiguity in C++. The ANSI X3J16 base documents do not state what the language is well enough that it can be implemented consistently. Nor do the base documents adequately describe any strategy or means by which C++ can be parsed. // it does To demonstrate this point, an appendix of simple programs, some taken directly from the manual, were presented to all the commercially available C++ processors with widely varying interpretations.

Doesn't prove the point

This paper requests clarification of syntactic ambiguities, and the correct meaning of C++ constructs. C++ is a very complex language syntactically. There are a number of serious ambiguities present in the language. Some of these ambiguities are not addressed at all in the base documents, others are treated inconsistently.

Based on the text of the base documents, and the additional commentary available in the C++ Annotated Reference Manual, it is not clear how sophisticated a parser must be to properly parse C++. It is difficult to determine where sophistication is required to implement language features, and where it seems to be required because the language has been inconclusively described.

One further complication is that the base documents are not entirely in agreement. Policies in C++ conflict directly with policies in ANSI C. Furthermore, various difficulties in ANSI C have been treated with less than full respect by C and C++ implementors alike.

As an example of how difficult it is to implement the current C++ language specification, the appendix contains simple code fragments that commercially available C++ implementations, advertising themselves in conformity with 2.0 or 2.1, do not implement correctly, or consistently. I welcome comparisons with other compilers, and will gladly provide machine readable versions of the test programs on request.

1. Declaration vs Expression Ambiguity

Section 6.8 of the ARM "Ambiguity Resolution" addresses expression-declaration ambiguity. The best-known case of this ambiguity the similarity between function-like casts and declarations with parenthesized declarators.

```
typedef int t;
t (x);
```

This is considered equivalent to

```
typedef int t;
t x;
```

The declaration of x is not to be misparsed as a cast. It is not the same as

```
(t) x;                  // line 2.  Misparsed.
```

This case is discussed directly in section 6.8 of the C++ Reference Manual.

> There is an ambiguity in the grammar involving **expression-statements** and **declarations.** An expression-statement with an explicitly type conversion (5.2.3) as its leftmost sub-expression can be indistinguishable from a declaration where the first declarator starts with a (. In those cases, the statement is a declaration.
>
> To disambiguate, the whole **statement** may have to be examined to determine if it is an **expression-statement** or **declaration.** This disambiguates many examples. For example, assume T is a **simple-type-name:**

```
T (*d) (double(3)); //expression-statement
T (a);              // declaration
```

> The disambiguation is purely syntactic; that is, the meaning of the names, beyond whether they are type-names or not, is not used in the disambiguation.

Supplementary commentary in the Annotated Reference Manual states:

> The general cases cannot be resolved without backtracking, nested grammars or similar "advanced" parsing strategies. In particular, the lookahead needed to disambiguate this case is not limited.
>
> n In a parser with backtracking the disambiguating rule can be stated very simply:
> [1] If it looks like a declaration, it is; otherwise
> [2] If it looks like an expression, it is; otherwise

I did not

[3] It is a syntax error.

Defining the language by example is extremely problematic from the perspective of formal language specification. It is certainly a radical departure from modern ANSI standard operating procedure, and deviates wildly from the customs of the international community. The characterization of legal language constructs in terms of non-standard parsing algorithms is also very questionable for a language standard.

Just as importantly, humans find it difficult to read and write programs with the above kind of specification. As an example of the difficulty, consider that in the canonical examples given in section 6.8 at least one of them is arguable by the very criteria given.

```
T(*d) (double(3));  // expression-statement
```

Yet, wouldn't it be more in keeping with the above syntactic rules that this should be a declaration of a pointer to an object T that is initialized with a double(3)? How would you know which one it is?

bug

Of the other canonical examples listed in section 6.8 of the C++ reference manual, several of those are consistently mistreated by commercial implementations. These discrepancies are noted in the appendix. The fact that conscientious implementors have had such difficulty deriving a consistent (much less conforming) interpretation suggests that the specification itself may lie at the root of the problem.

If a standard specification for C++ is to give users a fighting chance to write portable, understandable, and maintainable programs, at the very least this standard must define what is legal, and what it means.

While it might be argued that such ambiguities are obscure and a programmer may be able to rewrite his code so it is clearer, it is imperative that the language standard define the language so that disputes of what is legal and what isn't are rare and of straightforward resolution.

Posing the "how do you tell the difference between an expression and a declaration" question another way, what are the limits to which the compiler must go to resolve difficult cases? What are the limiting cases for deciding when something "can't possibly be a declaration?"

2. Declaration vs Declaration ambiguity

Section 6.8 does not mention ambiguities between declarations. To see how declaration vs declaration ambiguity can arise, bear in mind the text from sect 9.1 of the ARM.

> A class declaration introduces the class name into the scope where it is declared and hides any class, object, function, etc., of that name in an enclosing scope. If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared the class can be referred to only using an elaborated type specifier.

Yet this requires better coverage, as the following problems will show.

2.1 function prototype vs parenthesized declarator

Consider the following fragment:

```
typedef int t;
class x {
      int i;
      };

x (t);          // object or function?
```

Does the declaration of `x (t)` redefine a function `x` taking a `t` as an argument, or does it declare an object `t` of type `x`?

To be specific, is `x(t)` equivalent to

```
int x(int dummy);    // declaration of function "x"
```

or is it equivalent to

```
class x t;           // illegal redefinition of t
```

According to limitations on the redefinition of typedefnames at the same scope at which they are defined, this should be a function declaration. In particular, Section 7.1.3 states

> A typedef may not re-define a name of a type declared in the same scope to refer to a different type.

Unfortunately, this is more problematic at a local scope.

```
typedef int t;
class x {int i; };

main () {
      x(t);
      }
```

Now `x(t);` may be either a declaration of an object `t` of type `x`, or a function named `x` taking an argument of type `t`. Some implementations make this an object of type `t`. Yet, to be consistent, shouldn't it mean the same thing at local scope as it does at global scope?

Note that this ambiguity is independent of any anomalies that may occur if there are conversions in the class `x`, or any other semantic complications.

Another observation here is that the ANSI C grammar explicitly disallows declarations without declaration specifiers, so the C++ syntax is automatically at odds with ANSI C. In ANSI C, only function definitions allow the omission of the return type; function declarations without a return type are illegal. This issue is discussed more fully in section 4.1 of this paper.

redundancy relative to C, Does ANSI C compiler actually disallow ; f(int);

Removing the offending extension does not automatically fix the problem however. For example, how should the following be treated:

```
struct x { };
typedef int t;
main() {
     extern x(t);
     }
```

Does the local declaration declare a function (taking a `t`) or an object named `t`?

2.2 Parenthesized Initializer vs Function Prototype

Another intra-declaration ambiguity is between a parenthesized initializer, and a function prototype. Consider the following fragment:

```
class X {
     public:
     X(int,int,float);
     };

typedef int t;
typedef float f;
int x=30, y=40, z=50;

X foo(10,10,1.0);              // Object instance
X Going(t(x),t(y),1.0);        // Object instance
X Gone(t(x),t(x),f(x));        // Object instance
X FarGone(t(x),t(y),f(z));     // Function declaration
   // i.e. FarGone(int dummy1, int dummy2, float dummy3) or
   //       FarGone((int)x, (int)y, (int)z);
```

What any of these mean cannot be determined until after reading the last two characters on the line.

In particular the resolution of the whole statement depends on what `t(x)` is; it could be an abstract declarator of type `t` with dummy argument `x`, or it could be a cast of the global variable `x` to a `t`. Until each declarator or initializer is resolved, it's not clear whether `Going` is a function returning an `X` or an instance of `X` initialized according to the public constructor.

Note that `Gone` must be an object declaration because the prototype scope (x3j11 3.1.2.1) introduced by the function declaration dictates that the reuse of the same name in the same scope would be an illegal redeclaration. Since it cannot be a legal function declaration, it must instantiate an object.

At what point can the compiler decide whether something is an expression, a declaration, or an error? Must it make these heroic efforts to determine even simple cases? What are the implementation limits? And what about the user? We provide syntax to avoid these problems.

For the treatment of a related issue, see section 4.3, ahead.

2.3 Scope of declarators

At what point does a declarator come into scope? Does it come into scope before the initializer is parsed, or does it not come into scope until the last parenthesis is parsed?

Making declarators come into scope at the time the initializer is parsed, would be consistent with ANSI C, allowing

```
int x=sizeof(x);
```

But what about in parenthesized initializers?

```
int x(sizeof(x));
```

The reuse of the class name as an identifier poses problems:

```
class C {
          public:
          C(class C&);
          };

C C(C);                  // ??
     // ``class C C=C;''
     //    or the declaration of a function
     // ``extern class C C(class C dummy);''
```

Is C an object declaration, initialized with itself? Or is the second C still a type in scope as a type, making the whole thing a function declaration?

The behavior of typenames reused at a more local scope as variables introduces other difficulties. At what point do outer scope type definitions go out of scope?

Consider the following:

```
typedef int t;
foo) {
   int t=(sizeof(t));     // Which t?
   }
```

Based on this example, it might seem wise to make the name come into scope at the parenthesis. On the other hand, consider a more involved case

```
int T=100;
int x=10;

main() {
struct T {
          int x;
          T(int);
```

```
            T(T&);
            T operator*(int)
               {printf("%d\n", i); return i * x;};
            } T(T * x);     // object or function?
T.x;
}
```

The net effect of the final declaration of T might be an object initialized with an integer expression, or it might be a function taking a T *. It could also be a T initialized by multiplying a T by an int. Alternatively, it could be a T initialized with itself times an int, using the overloaded operator that the class T supplies.

A strict reading of the base documents, suggests a contradiction. In particular, it would seem that the items in a declarator list must be both typenames and identifiers in scope, at the same time, and in the same place. Consider the following:

> If an outer declaration of a lexically identical identifier exists in the same name space, it is hidden until the current scope terminates, after which it becomes visible.
>
> 3.1.2.1 (.35) x3j11
>
> ... Any other identifier has scope that begins just after the completion of its declarator.
>
> 3.1.2.1 (.41) x3j11

In general, how should the coming into scope of a declarator work in the case of self-initialization?

For example, does

```
class B { B(&B); };
B B(B);
```

define an object initialized with itself, or a function taking a B?

3. What is the Order of Parsing For Inline Member Functions?

According to the C++ reference manual (9.3.2)

> Defining a function within a class declaration is equivalent to declaring it inline and defining it immediately after the class declaration; this rewriting is considered to be done after the preprocessing but before syntax analysis and type checking.

This allows a member function declared inline to have access to the entire data definition of the class, and have access to all the member functions, and presumably, (although it isn't stated) any types that may be declared in the class definition.

On the other hand, consider this passage: (9.9 in the 2.1 documentation, 9.3.2 in the 2.0 doc)

> A class-name or a typedef-name or the name of a constant used in a type name may not be redefined in a class declaration after being used in the class declaration, nor may a name that is not a class-name or a typedef-name be redefined to a class-name or a typedef-name in a class declaration after being used in the class declaration.

Thus it would seem that a member function cannot be parsed until the entire class definition has been seen. On the other hand, the class definition cannot be parsed until all uses of names as typedefs is known. What is the availability of typedef names in a member function, and what is a parsing order that will always correctly parse a class definition?

In the following fragment, what should f and g do?

```
class X {
     f() { return sizeof(T); }
     g() { T x; }
     typedef int T;
     };
```

Submitting the following example taken from 9.9 to the extant implementations of C++ sadly produced widely varying results. The following example has appeared unchanged in both the C++ 2.0 and C++ 2.1 manuals.

```
typedef int c;
enum { i = 1 };

class X {
     char v[i];
     int f() { return sizeof(c); }
     char c;  //error: typedef name redefined after use
     enum {i = 2 };  //error: 'i' redefined after v[i]
     };

typedef char *T;

struct Y {
     T a;
     typedef long T;      // error: T already used
     t b;
     };
```

Based on the material in the C++ manual, another possible interpretation of the visibility of names as typedef might be to determine that a name is a typedef at the point it is first declared or used as one. Subsequently, the actual meaning of the typedef is bound as late as possible. (This is not a proposal, just a recognition of a possible interpretation.) For example,

```
typedef float T;
class x {
          bar() {T x;}
          typedef char T;
          }
```

This might mean that the actual type of local x in function bar is a character, not a float.

3.1 Parsing nested classes and friend declarations

In the case of friends in nested classes, does the name of a friend look in or out for it's resolution? For example

```
class W { f(); };

class X {
          friend W::f();
          friend W;
          class W {
                    f();
                    };
          }
```

Which class W is the friend class? Which W::f is a friend function?

While nested classes were added to C++ as a technical correction between 2.0 and 2.1 their semantics are still unclear.

4. Matters of Improved Formalism

There are a number of areas where the language specification could improve its treatment by employing more rigorous formalism. These parsing problems do not pose definite parsing ambiguities (that I know of!), unless otherwise noted.

4.1 What is a declaration?

According to the C++ reference manual it is

```
decl_specifiers[opt] declarator-list;
```

According to X3J11 it is

```
decl_specifiers init-declarator-list;
```

In ANSI C, the following is a syntax error because type specifiers cannot be omitted in the redeclaration of a typedef name. (X3J11 3.5.6 line 15)

```
typedef int Pc;
foo() { const Pc; }
```

But according to the C++ reference manual (Section 7.1), decl_specifiers are parsed as follows.

> The longest sequence of decl-specifiers that could possibly be a typename is taken as the decl-specifiers of a declaration. The sequence must be self-consistent as described below.
>
> For example

```
typedef char *Pc;
foo() { static Pc;        /* error */ }
```

However given the C++ definition it seems odd that the C++ manual doesn't consider

```
static Pc;
```

correct: treat this by saying that the longest possible sequence of decl-specifiers stops with static.

4.1.1 Boundaries Between Legal and Illegal Declarations

Note that the following code fragments are deemed illegal by X3J11, by a combination of 3.5

> A declaration shall declare at least a declarator, a tag or the members of an enumeration.

and 3.5.6

> If the identifier is redeclared at an inner scope ... the type specifiers shall not be omitted.

```
{ /* 1 */
typedef int t;
{ const *t;}
}

{ /* 2 */
typedef int t;
{ const x, t; }
}

{ /* 3 */
typedef int t;
{ const t; }
}
```

The first two fragments are innocuous, since the decl_specifier is delimited by another token, and its correct interpretation as a declaration is forced. Even simple-minded parsing strategies handle these statements as correct declarations; in fact, it takes extra work to determine the illegal cases, work which many compilers do not bother to do.

While the rules cover case 3, above, they unnecessarily nullify cases 1 and 2. By carefully formulating the grammar, both of these prose passages from the standard could have been avoided, while making the implementation easier and the language more uniform.

4.2 Name visibility in constructor-initializers

A constructor initializer list allows the initialization of both member class variables, and baseclass initializers. The syntax is necessary for both cases: there is no other way to specify arguments to base class constructors, and there is no other way to initialize const data members.

Yet there is no stated preference for visibility. Consider the case where a class definition has base class and data member of the same name.

```
class x {
      int i;
      x(int);
      };
class y : public x {
      const int x;
      y(int);
      };
y::y(int) : x(10) { }
```

Since the only time the baseclass name is in scope as such is in the constructor initializer list, and the only time a const member is initializable is in the same place, there is a conflict.

4.3 Artifice in abstract declarators

Consider the following from the ANSI C specification:

> In a parameter declaration, a single typedef name in parenthesis is taken to be an abstract declarator that specifies a function with single parameter, not as redundant parentheses around the identifier for a declarator.
>
> (X3J11 3.5.4.3)

This ruling covers the following case

```
typedef int t;
void foo(int(t));
```

And rules that foo is a function taking a pointer to a "function-returning-int" as a parameter. It is not the declaration of a function taking an int whose name is t. Yet it probably does not cover all the intended cases. Consider the following direct analog:

```
typedef int t;
void foo(int (t[]));
```

No doubt this is assumed to be the same sort of case involving a function taking a pointer to a "function-returning-int" as a parameter, but taking an array of t whereas the former function took a single t. Yet the ANSI C specification does not cover this case.

4.4 Conformance and Grammar

Experience with ANSI C seems to bear out the fact that rules posed in the grammar tend to be implemented more uniformly than rules that are not. For example, fewer compilers seem to flag vacuous declarations like

```
int ;           /* illegal declaration */
```

than flag degenerate function declarations like

```
f();            /* illegal declaration */
```

at global scope. Both are illegal. But one is tolerated by most compilers; while one isn't. The latter violates ANSI C syntax, and is almost always detected.

Other rules that are stated in prose tend to be ignored altogether -- as evidence of this consider that

```
typedef int t;
foo() { const *t; }
```

is illegal according to the rules, as is

```
typedef int t;
foo() { const a, t; }
```

but some compilers not detect any errors here.

In conclusion, language definitions that employ standard formal techniques are more reliable. For example, parser generators can detect ambiguities that are difficult to detect by hand. And reliable consistent parsers can be constructed by independent implementors, allowing for uniform and consistent user experience.

Clearly languages with "advanced" parser technology make construction, modification, and transportability of programs harder. What is difficult for the compiler to figure out is also difficult for human programmers to figure out. Therefore, ambiguous specifications impose widely varying standards of conformance on the C++ developer in return for language features that are harder to use and understand.

```
' p0.cxx */
/pedef int t;
lass x { public: int i; };

:t);                    // 5. Illegal redecl. of t or questionable func. proto.?
it x(t);                // 6.

ain() {
        x(10);          // 9.
        }

(t i) {                 // 12.
        class x xxx;    // 13. note: typename is hidden, ''class'' req.
        xxx.i = 10;
        return 10;


/ -- p0.cxx --
/
/ #a is not a legal redeclaration of t as an x, since at the same scope.
/ According to ANSI X3J16, it is not a legal function declaration.
/ It is legal as a function declaration in C++, however.
/
/ Cfront 2.0:  No errors
/ tc++:        line 5, multiple declaration of t
/ Zortech:     line 6, 12 syntax error... Cascading errors
/ cfront 2.1:  No errors
/ g++ 1.37.2.1:line 5 "parse error before ';'"
/ g++ 1.37.2.1:line 12 "redefinition of 'int x(int)'"
/
```

```
    1.cxx */
typedef int t;
class x { public: int i; };

main() {
        x(t);                   // 6. redeclaration of t, or function call?
        int x(t);
        x(10);
        }

x(t i) {
        class x xxx;            // 12. note: typename is hidden, ''class'' req.
        xxx.i = 10;
        return 10;
}

// -- p1.cxx --
//
// Compare with p0.cxx.  The only difference is that the declarations
//  of t are at a local scope.
//
// cfront 2.0: syntax error line 7
// tc++:        line 6 "improper use of typedef."  didn't handle function
//              after that, "right paren expected."
// zortech:     line 7, "param list out of context"
// cfront 2.1:  line 6,  "syntax error"
// g++ 1.37.2.1:line 6, "parse error before ')'"
//
```

```
' p2.cxx */
.ass X {
        public:
        X(int, int, float);
        };

.pedef int t;
.pedef float f;
it x=30, y=40, z=50;

 foo(10,10,1.0);                        // 11. clear object
 Going(t(x), t(y), 1.0);                // 12. object
 Gone(t(x), t(y), f(z));                // 13. function prototype
 FarGone(t(x), t(x), f(x));             // 14. Can't be a prototype!

ain() {
        Gone(10,10,1.0);
        }

/ -- p2.cxx --
/
/ Declaration of objects (with parened initializers) or function prototypes?
/
/ 14. can't be the declaration of a function prototype because dummy variable
/     names must be unique; otherwise they would be illegal redeclarations.
/
/ cfront 2.0:  line 12 "argument declaration error", thinks rest are functions
/ tcc++:       line 17 "call of non-function in main"  Seems to think
/              the function prototypes are objects.
/ zortech:     line 12 "syntax error"
/ cfront 2.1:  line 12, "argument declaration syntax" and 2 syntax errors
/ g++ 1.37.2.1:line 17, "called object is not a function"
/
```

```
/* p3.cxx */
typedef int t;

main() {
t(x);           // 5.
}

// -- p3.cxx --
//
// Example from Section 6.8
//
// cfront 2.0:  line 5, x undefined
// tc++:        line 5, undefined symbol x in function main
// Zortech:
// cfront 2.1:  line 5, x undefined
// g++ 1.37.2.1:line 5, x undeclared
```

```
* p4.cxx */
ypedef int t;

ain() {
(*d)(double(3));        // 5.


/ -- p4.cxx --
/
/ Example from section 6.8
/
/ Cfront 2.0:  line 5, error d undefined
/ Tc++:        line 5, syntax error
/ Zortech:
/ cfront 2.1:  line 5, error d undefined
/ g++ 1.37.2.1:line 5, d undeclared
```

```
   5.cxx */
lass T {
        public:
        T(int,int);
        };

main() {
int h=10;
T(g)(h,2);      // 9.
}

// -- p5.cxx--
//
// Example from section 6.8
//
// cfront 2.0:  line 9, argument 2 of type int expected for T::T()
// cfront 2.0:  line 9, bad operand type, struct T for ()
// Tc++:        line 9, undefined symbol g
// Zortech:
// g++ 1.37.2.1:line 9, warning implicit declaration of function g
// g++ 1.37.2.1:line 9, error too few arguments for constructor 'T'
// cfront 2.1:  line 9, undefined function g called
// cfront 2.1:  line 9, argument 2 of type int expected for T::T()
```

```
/* init0.cxx */
main() {
        int i=sizeof(i);        // 3.
        int j(sizeof(j));       // 4.


//   -- init0.cxx --
//
// Clearly legal in 2.1 see section 8.4.
// According to the grammar, and since not explicitly
// disallowed, legal in 2.0 also.
//
// cfront 2.0:  line 4, "argument type expected for j"
// tc++:        line 4, "expected ) in function main"
// zortech:     line 4, "syntax error"
// cfront 2.1:  line 4, "argument type expected for j()"
// g++ 1.37.2.1:line 4, "'j' undeclared", "bad parameter list"
```

```
/* init1.cxx */

struct C {
        C(class C&);
        C();
        };
struct D {
        D(class D&);
        D();
        };
struct E {
        E(class E&);
        E();
        };
typedef int t;

C f(t);         // 17. #a. Function prototype?  (Seems forced)
C C(t);         // 18. #b. Function prototype? (Seems forced)
D D(D);         // 19. #c. Function or self-initialized object??
E e;
E E(e);         // 21. #d. Object?  (Seems forced)

class C C(C&);  // 23. Function...
class D D(D&);  // 24. Function...

main() {
        class C c;
        class D d;

        f(10);          // 30. Function call.
        C(c);           // 31. #e. Function call or re-declaration?
        D(d);           // 32. #f. Function call or re-declaration?
        }

// -- init1.cxx --
//
// Notice how the 2.0 scoping rules prohibit a class with a constructor
//   from being confused with a function of the same name.
//
// cfront 2.0:  Line 18, 19, 21, 23, 24
// cfront 2.0:    "-Redefined both as class w/constr & identifier"
// cfront 2.0:  Line 24
// cfront 2.0:    "The overloading mechanism cannot tell void(D) from void(D&)"
// cfront 2.0:  Line 32 "two exact matches for D()"
//
// tc++:        Line 23, "Type mismatch in redeclaration of 'C'"
// tc++:        Line 23, "Operand expected"
// tc++:        Line 24, ... same ...
// tc++:        Line 31, "Call of non-function"
//
// zortech:     Line 4, 8, 18, "syntax error"
// zortech:     Line 18, "No constructor allowed for class C"
// zortech:     Line 19, "No constructor allowed for class D"
//
// cfront 2.1:  same as for 2.0
// g++ 1.37.2.1:Line 23,24, "parse error before ')'"
```

```
* init2.cxx */
truct C {
   int x;
   };

oid o1() {
   C(C);       // 7. C is an object of type C
   }

oid o2() {
   C(*C);      // 11. C is a ptr to a C
   }

oid o3() {
   C(*C)();    // 15. C is a ptr to a func returning a C
   }

oid o4() {
   C(*C)(struct C);    // 19. C is a ptr to a function expecting a C as an arg
   }

oid o5() {
   C(*C)(C);           // 23. Same, depending on when C loses defin. as type
   }

xtern printf(const char *, ...);

oid c() { printf("easy c\n"); }

oid (*C(void (*v)()))() { printf("Wierd C\n"); return c;}      // 30.
oid (*C())();           /* 31. Declared w/out parameter */
oid (*C(void *))();     /* 32. Declared with parameter  */

oid f1() {
   C(C);       // 35. Call C pass C as an argument
   }

oid f2() {
   C(*C);      // 39. Call C pass what it points to
   }

oid f3() {
   C(*C)();    // 43. Call C, call what comes back
   }

oid f4() {
   C(*C)(C);   // 47. Call C, call what comes back passing C
   }

in() {
intf("f1\n"); f1();
intf("f2\n"); f2();
intf("f3\n"); f3();
intf("f4\n"); f4();
turn 0;


   -- init2.cxx --
```

```
   See init2.c for ansi C version of same code.

   cfront 2.0: line 7, 11, 15, 19, "syntax error"
// cfront 2.0: line 32, 36, 40, 44, "bad operand type"
// cfront 2.0: ... Bad argument list, non-ptr dereferenced, etc
// cfront 2.0: errors on 7, 11, 15, 19, 27, 28, 29, 32, 36, 40, 44

// tc++:         line 7, 11, 15 "call of non-function"
//               line 19 "expression expected"
//               line 27 seems to be lost hereafter

// zortech:      line 20, 44 "syntax error" 20, 44
//
// cfront 2.1:  line 7, 11, 15, 19, 23, "syntax error"
// cfront 2.1: ... Bad argument list, non-ptr dereferenced, etc
// cfront 2.1:  errors on line 35, 39, 43, 47,
//
// g++ 1.37.2.1:line 7, 11, 15, 19, 23 "parse error before ')'"
// g++ 1.37.2.1:line 30, "'v' undeclared, outside of functions"
// g++ 1.37.2.1:line 35, "insufficient type information in parameter list"
// g++ 1.37.2.1:lin4 39, 43, 47 "invalid type argument of 'unary *'"
//
```

```
/* init3.cxx */

ass x {
        public:
        int i;
        x(int);
        x();                    // 7. Try this...
        };

::x(int p) { i = p; }

ass y : public x {
        const int x;
        public:
        y(float f);
        };

::y(float f) : x(0)             // 18. Which x does this refer to?
  { }

in() {
        y Ay(1.0);
        }


/ --- init3.cxx ---
/
/ If there is a base class and a member variable of the same name,
/  which gets precedence?
/
/ cfront 2.0:   Line 18 "Argument 1 of type int expected"
/
/ tc++:         Line 18 "Cannot find x::x() to initialize"
/
/ zortech:      Line 21 "Syntax error. not found or ambig reference to function"
/
/ cfront 2.1:   same as for 2.0
/
/ g++ 1.37.2.1:no errors reported
/
```

```
/* init4.cxx */

int T=10;
int x=20;

extern "C" printf(const char *, ...);

void foo() {
struct T {
                int x;
                T operator *(int i) { printf("%d\n", i); return i * x; }
                T(int);
                T(T&);
        } T(T * x);     // create T(10*20) or declare function with T(T*) arg

printf("%d\n", T.x);
}

main() {
foo();
}

//  -- init4.cxx --
//  This should probably be the declaration of a T initialized with 10*20;
//  Or it could be a T initialized with a T * i, or it could be
//  a declaration of a function returning a T taking a T * as an argument.
//
//  But it seems indeterminate to me.
//
// cfront 2.0:  Line 16, "Object or object pointer missing for member x"
// cfront 2.0:  Line 16, ". used for qualification, please use ::"
//  (apparently thinks T is a function)
//
// tc++:        Line 16, "Unreachable code"
// tc++:        Line 16, "Left side must be a structure"
//
// cfront 2.1:  Same as 2.0
//
// g++ 1.37.2.1: Line 16, "Parse error before '.'"
```

```
/* order0.cxx */

class X {
        f() { return sizeof(T); }       // 4.
        g() { T x; }                    // 5.
        typedef int T;                  // 6.
        };

// -- order0.cxx --
//
// cfront 2.0:  T x; is not a typename
// tc++:        No errors reported
// zortech:     No errors reported
// cfront 2.1:  No errors reported
// g++ 1.37.2.1:No errors reported
//
```



```
/* order1.cxx */
typedef int c;
enum { i = 1 };

class X {
        public:
        char v[i];
        int f() { return sizeof(c); }
        char c;                 // 9.  error -- redefinition
        enum { i = 2 };         // 10. error -- redefinition
        X();
        };

typedef char *T;                // 14.

struct Y {
    T a;
    typedef long T;             // 18. error -- T already used
    T b;                        // 19.
    Y();                        // 20.
    };

extern printf(const char *, ...);

main() {
X x;
Y y;
printf("%d\n", x.f());
}

// -- order1.cxx --
//
// Taken from the manual.  Clearly valid for both 2.0 and 2.1.  Lines
// marked "error" were marked as such in the manual, and  should
// produce errors from the compiler.
//
// This is the example in section 9.9 with only constructors and a main
// added.
//
// cfront 2.0:  Line 18.  T redeclared. print(sizeof(int)).
// tc++:        Line 18.  "multiple declaration for 'T'"  print(sizeof(char));
// zortech:     "No tag for struct or enum"
//              Line 18.  "'T' previously declared as something else"
//
// cfront 2.1:  line 18, "T declared as char* and long"
// g++ 1.37.2.1:no errors reported
//
```

```
* order2.cxx */
ypedef int t;
lass x {
   t ttt;
   int t;              // error: illegal reuse of name
   x();
   };

::x() {
   int i;
   t = 100;
   i = t;
   ttt = i;
   }

ain() {
eturn 0;


/ -- order2.cxx --
/
/ cfront 2.0:  Detects no errors.
/ tc++:        Detects no errors.
/ zortech:     Detects no errors.
/ cfront 2.1:  Detects no errors.
/ g++ 1.37.2.1:Detects no errors.
```



```
/* order3.cxx */
typedef int t;
class x {
    t ttt;
    int t;              // 5. error: illegal reuse of name
    typedef int t;      // 6. error: illegal reuse of reuse
    x();
    };

x::x() {
        t = 100;        // 11. error: t should be visible only as typedef
    }

main() {
return 0;
}

// -- order3.cxx --
//
// cfront 2.0:  Detects no errors
// tc++:        Detects no errors
// zortech:     Detects no errors
// cfront 2.1:  Line 6, "nested typedef t redefines global t"
// cfront 2.1:  Line 6, "Can't recover"
// g++ 1.37.2.1:Line 6, "duplicate member 't"
// g++ 1.37.2.1:Line 11, "parse error before '='"
```