Context-free problems

The C++ (and C) grammar violates the requirement for a context-free grammar,since new context-dependent keywords are introduced into a program by typedef, namespace, class, enumeration and template declarations. When an identifier is encountered, semantic information is needed, since there is a context-dependency on type names and on template names. If the grammar defined by the standard is to be followed very closely, semantic information is apparently needed to classify identifiers into one of :

- class-name
- enum-name
- identifier
- namespace-alias
- original-namespace-name
- template-name
- typedef-name

A full semantic interpretation of a C++ program obviously requires a knowledge of the types. Unfortunately this information is also needed for a complete syntactic disambiguation of :

- declaration/expression ambiguity
- parenthesised-call/cast-parenthesis ambiguity
- parenthesised-binary/cast-unary ambiguity
- call/functional-cast ambiguity
- new placement/initializer ambiguity
- sizeof type/value ambiguity
- typeid type/value ambiguity
- template argument type/value ambiguity

The traditional parsing approach needs to resolve semantics during syntactic analysis and so encounters ambiguities that need type information.
Reference: Meta-compilation for C++

Declaration/Expression Ambiguity
https://www.ibm.com/docs/en/i/7.2?topic=statements-resolution-ambiguous-c-only
The C++ syntax does not disambiguate between expression statements and declaration statements. The ambiguity arises when an expression statement has a function-style cast as its leftmost subexpression. (Note that, because C does not support function-style casts, this ambiguity does not occur in C programs.) If the statement can be interpreted both as a declaration and as an expression, the statement is interpreted as a declaration statement.

Reference:
https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL53-CPP.+Do+not+write+syntactically+ambiguous+declarations
Noncompliant Code Example:
In this noncompliant code example, an anonymous local variable of type std::unique_lock is expected to lock and unlock the mutex m by virtue of RAII. However, the declaration is syntactically ambiguous as it can be interpreted as declaring an anonymous object and calling its single-argument converting constructor or interpreted as declaring an object named m and default constructing it. The syntax used in this example defines the latter instead of the former, and so the mutex object is never locked.

```cpp
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex>(m);
  shared_resource += 42;
}
```

Compliant Solution:
In this compliant solution, the lock object is given an identifier (other than m) and the proper converting constructor is called.

```cpp
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex> lock(m);
  shared_resource += 42;
}
```

The following noncompliant code example demonstrates a vexing parse. The declaration Gadget g(Widget(i)); is not parsed as declaring a Gadget object with a single argument. It is instead parsed as a function declaration with a redundant set of parentheses around a parameter.

Noncompliant Code Example:
```
#include <iostream>

struct Widget {
  explicit Widget(int i) { std::cout << "Widget constructed" << std::endl;
}
};

struct Gadget {
  explicit Gadget(Widget wid) { std::cout << "Gadget constructed" <<
std::endl; }
};

void f() {
  int i = 3;
  Gadget g(Widget(i));
  std::cout << i << std::endl;
}
```
Parentheses around parameter names are optional, so the following is a semantically identical spelling of the declaration.
```
Gadget g(Widget i);
```
Clang: compiled with 0 error,
```
warning: parentheses were disambiguated as a function declaration
      [-Wvexing-parse]
  Gadget g(Widget(i));
          ^~~~~~~~~~~
mvp.cpp:16:12: note: add a pair of parentheses to declare a variable
  Gadget g(Widget(i));
           ^
           (         )
1 warning generated.
```
g++: compiled with 0 error, 0 warnings

As a result, this program is well-formed and prints only 3 as output because no Gadget or Widget objects are constructed.

Unambiguous Solution:

This solution demonstrates two equally compliant ways to write the declaration of g. The first declaration, g1, uses an extra set of parentheses around the argument to the constructor call, forcing the compiler to parse it as a local variable declaration of type

Gadget instead of as a function declaration. The second declaration, g2, uses direct initialization to similar effect.

```
#include <iostream>

struct Widget {
  explicit Widget(int i) { std::cout << "Widget constructed" << std::endl;
}
};

struct Gadget {
  explicit Gadget(Widget wid) { std::cout << "Gadget constructed" <<
std::endl; }
};

void f() {
  int i = 3;
  Gadget g1((Widget(i))); // Use extra parentheses
  Gadget g2{Widget(i)}; // Use direct initialization
  std::cout << i << std::endl;
}
```

Running this program produces the expected output.

```
Widget constructed
Gadget constructed
Widget constructed
Gadget constructed
3
```

Parenthesised-call/ Cast-parenthesis Ambiguity

A full semantic interpretation of a C++ program obviously requires a knowledge of the types. Unfortunately this information is also needed for a correct syntactic interpretation of an expression using a C cast followed by a unary operator or call.

```
(T)(5) // This is a cast if T is a type
(t)(5) // This is a function call if t is not a type
```

```
#include <stdio.h>
int t(float f){
    return int(f);
}
```

```
int main(int argc, char const *argv[])
{
    typedef int T;
    T num1 = (T)(8.3);
    printf("%d\n", num1);
    int num2 = (t)(5.2);
    printf("%d\n", num2);
    return 0;
}
```
Output:
```
8
5
```

Parenthesised-binary/Cast-unary Ambiguity

```
(T)-5  // This is a cast if T is a type
(t)-5  // This is a subtraction if t is not a type
```

```
#include <stdio.h>

int main(int argc, char const *argv[])
{
    typedef int T;
    float t=2.1;
    int a=(T)-5;
    printf("%d\n", a);
    int b=(t)-5;
    printf("%d\n", b);
    return 0;
}
```
Output:
```
-5
-2
```

Direct base class inaccessible in derived due to ambiguity

1. Prob 1:

https://stackoverflow.com/questions/30246098/warning-direct-base-class-inaccessible-in-derived-due-to-ambiguity-is-this-ser?rq=1

2. Base & derived class ambiguities => https://stackoverflow.com/questions/4118412/inaccessible-direct-base-caused-by-multiple-inheritance

Ambiguous Code

```
#include <string>
#include <unordered_map>

class Spell {
    protected:
        struct Exemplar {};
        Spell() = default;
        Spell (Exemplar, const std::string&);
};

class SpellFromScroll : virtual public Spell {
    private:
        static std::unordered_map<std::string, SpellFromScroll*>
prototypesMap;
    public:
        static void insertInPrototypesMap (const std::string& tag,
SpellFromScroll* spell) {
            prototypesMap.emplace (tag, spell);
        }
        template <typename T> static SpellFromScroll* createFromSpell
(T*);
};
std::unordered_map<std::string, SpellFromScroll*>
SpellFromScroll::prototypesMap;

class SpellWithTargets : virtual public Spell {};  // *** Note:
virtual

class Sleep : public SpellWithTargets {
    private:
        static const Sleep prototype;
    public:
        static std::string spellName() {return "Sleep";}
    private:
        Sleep (Exemplar e) : Spell (e, spellName()) {}
};
const Sleep Sleep::prototype (Exemplar{});

template <typename T>
class ScrollSpell : /*virtual*/ public T, public SpellFromScroll {};

Spell::Spell (Exemplar, const std::string& spellName) {
    // Ambiguity warning!
    SpellFromScroll::insertInPrototypesMap (spellName,
SpellFromScroll::createFromSpell(this));
}

template <typename T>
SpellFromScroll* SpellFromScroll::createFromSpell (T*) {
    return new ScrollSpell<T>;
}

int main() {}
```

clang :

```
 warning: direct base 'Spell' is inaccessible due to
      ambiguity:
    class ScrollSpell<class Spell> -> class Spell
    class ScrollSpell<class Spell> -> class SpellFromScroll -> class
Spell [-Winaccessible-base]
class ScrollSpell : /*virtual* public T, public SpellFromScroll {};
                                ^~~~~~~~
derived_class_ambiguities_1_amb.cpp:44:16: note: in instantiation of
template class
      'ScrollSpell<Spell>' requested here
    return new ScrollSpell<T>;
               ^
derived_class_ambiguities_1_amb.cpp:39:73: note: in instantiation of
function template
```

```
      specialization 'SpellFromScroll::createFromSpell<Spell>'
requested here
    SpellFromScroll::insertInPrototypesMap (spellName,
SpellFromScroll::createFromSpell(this));

^
1 warning generated.
```

g++ :

```
derived_class_ambiguities_1_amb.cpp: In instantiation of 'class
ScrollSpell<Spell>':
derived_class_ambiguities_1_amb.cpp:44:12:   required from 'static
SpellFromScroll* SpellFromScroll::createFromSpell(T*) [with T =
Spell]'
derived_class_ambiguities_1_amb.cpp:39:93:   required from here
derived_class_ambiguities_1_amb.cpp:35:7: warning: direct base
'Spell' inaccessible in 'ScrollSpell<Spell>' due to ambiguity
 class ScrollSpell : /*virtual*/ public T, public SpellFromScroll {};
```

Using virtual keyword resolves this.

<u>Using the value of a function that has no return statement specified</u>

Ambiguous Code

```
int f(){
   int a = 10;
   int count = a * 10;
}

int main(int argc, char const *argv[])
{
   int count;
   count = f();
   return 0;
}
```

Clang :

```
no_return_statement_1_amb.cpp:4:1: warning: control reaches end of
non-void function
      [-Wreturn-type]
}
^
1 warning generated.
```
g++ : 0 error, 0 warnings. Compiled Successfully.

Unambiguous Code

```
int f(){
    int a = 10;
    int count = a * 10;
    return count;
}

int main(int argc, char const *argv[])
{
    int count;
    count = f();
    return 0;
}
```

Name Hiding

Suppose two subobjects named A and B both have a member name x. The member name x of subobject B hides the member name x of subobject A if A is a base class of B. The following example demonstrates this:

```
struct A {
  int x;
};

struct B: A {
  int x;
  void f() { x = 0; }
};

int main() {
  B b;
```

```
   b.f();
 }
```

The assignment x = 0 in function B::f() is not ambiguous because the declaration B::x has hidden A::x.
A base class declaration can be hidden along one path in the inheritance graph and not hidden along another path. The following example demonstrates this:

```
 struct A { int x; };
 struct B { int y; };
 struct C: A, virtual B { };
 struct D: A, virtual B {
   int x;
   int y;
 };
 struct E: C, D { };

 int main() {
   E e;
 //   e.x = 1;
   e.y = 2;
 }
```

The assignment e.x = 1 is ambiguous. The declaration D::x hides A::x along the path D::A::x, but it does not hide A::x along the path C::A::x. Therefore the variable x could refer to either D::x or A::x. The assignment e.y = 2 is not ambiguous. The declaration D::y hides B::y along both paths D::B::y and C::B::y because B is a virtual base class.

Ref-qualifiers on member functions

C++11 allows member functions to be overloaded on the value type of the object that will be used for this using a ref-qualifier. A ref-qualifier sits in the same position as a cv-qualifier and affects overload resolution depending on if the object for this is an lvalue or an rvalue:

```
#include <iostream>

struct Foo {
  void foo() & { std::cout << "lvalue" << std::endl; }
  void foo() && { std::cout << "rvalue" << std::endl; }
};
```

```
int main() {
  Foo foo;
  foo.foo(); // Prints "lvalue"
  Foo().foo(); // Prints "rvalue"
  return 0;
}
```

Pointer-to-member operators

Pointer-to-member operators let you describe a pointer to a certain member on any instance of a class. There are two pointer-to-member operators, .* for values and ->* for pointers:

```
#include <iostream>
using namespace std;

struct Test {
  int num;
  void func() {}
};

// Notice the extra "Test::" in the pointer type
int Test::*ptr_num = &Test::num;
void (Test::*ptr_func)() = &Test::func;

int main() {
  Test t;
  Test *pt = new Test;

  // Call the stored member function
  (t.*ptr_func)();
  (pt->*ptr_func)();

  // Set the variable in the stored member slot
  t.*ptr_num = 1;
  pt->*ptr_num = 2;

  delete pt;
  return 0;
}
```

This feature is actually really useful, particularly for writing libraries. For example, Boost::Python (a library for binding C++ to Python objects) uses member pointers to easily refer to members when wrapping objects:

```
#include <iostream>
#include <boost/python.hpp>
using namespace boost::python;

struct World {
  std::string msg;
  void greet() { std::cout << msg << std::endl; }
};

BOOST_PYTHON_MODULE(hello) {
  class_<World>("World")
    .def_readwrite("msg", &World::msg)
    .def("greet", &World::greet);
}
```

Keep in mind when using member function pointers that they are different from regular function pointers. Casting between a member function pointer and a regular function pointer will not work. For example, member functions in Microsoft's compilers use an optimized calling convention called thiscall that puts the this parameter in the ecx register, while normal functions use a calling convention that passes all arguments on the stack.

Also, member function pointers may be up to four times larger than regular pointers. The compiler may need to store the address of the function body, the offset to the correct base (multiple inheritance), the index of another offset in the vtable (virtual inheritance), and maybe even the offset of the vtable inside the object itself (for forward declared types).

```
#include <iostream>

struct A {};
struct B : virtual A {};
struct C {};
struct D : A, C {};
struct E;

int main() {
  std::cout << sizeof(void (A::*)()) << std::endl;
  std::cout << sizeof(void (B::*)()) << std::endl;
  std::cout << sizeof(void (D::*)()) << std::endl;
  std::cout << sizeof(void (E::*)()) << std::endl;
  return 0;
}

// 32-bit Visual C++ 2008:  A = 4, B = 8, D = 12, E = 16
// 32-bit GCC 4.2.1:        A = 8, B = 8, D = 8,  E = 8
// 32-bit Digital Mars C++: A = 4, B = 4, D = 4,  E = 4
```

All member function pointers in the Digital Mars compiler are the same size due to a clever design that generates "thunk" functions to apply the right offsets instead of storing the offsets in the pointer itself.

Static methods on instances

C++ lets you invoke static methods from an instance in addition to invoking them from the type. This lets you change an instance method to a static method without needing to update any call sites.

```
struct Foo {
  static void foo() {}
};

// These are equivalent
Foo::foo();
Foo().foo();
```

Overloading ++ and --

C++ is designed so the function name of custom operators is the operator symbol itself, which works fine in most cases. For example, the unary - and binary - operators (negation and subtraction) can be distinguished by the argument count. This doesn't work for the unary increment and decrement operators though since they both seem to need the exact same signature. The C++ language has an ugly hack to work around this: the postfix ++ and -- operators must take a dummy int argument as a flag for the compiler to know to make a postfix operator (and yes, only the type int works).

```
struct Number {
  Number &operator ++ (); // Generate a prefix ++ operator
  Number operator ++ (int); // Generate a postfix ++ operator
};
```

Functions as template parameters

It's well known that template parameters can be specific integers but they can also be specific functions. This lets the compiler inline calls to that specific function in the instantiated template code for more efficient execution. In the example below, the function memoize takes a function as a template parameter and only calls the function for new argument values (old argument values are remembered from the cache).

```
#include <map>

template <int (*f)(int)>
int memoize(int x) {
  static std::map<int, int> cache;
  std::map<int, int>::iterator y = cache.find(x);
  if (y != cache.end()) return y->second;
  return cache[x] = f(x);
}

int fib(int n) {
  if (n < 2) return n;
  return memoize<fib>(n - 1) + memoize<fib>(n - 2);
}
```

Template template parameters

Template parameters can actually have template parameters themselves. This allows you to pass templated types without template parameters when instantiating a template. Say we have the following code:

```
template <typename T>
struct Cache { ... };

template <typename T>
struct NetworkStore { ... };

template <typename T>
struct MemoryStore { ... };

template <typename Store, typename T>
struct CachedStore {
  Store store;
  Cache<T> cache;
};

CachedStore<NetworkStore<int>, int> a;
CachedStore<MemoryStore<int>, int> b;
```

CachedStore puts a cache that holds a certain data type in front of a store that stores the same data type. However, we must repeat the data type (int in the code above) when instantiating a CachedStore, once for the store itself and once for CachedStore, and there's no guarantee that the data types are consistent. We really want to just specify the data type once so we can enforce this invariant, but leaving off the type parameter list causes a compile error:

```
// These do not compile because NetworkStore and MemoryStore are missing type parameters
CachedStore<NetworkStore, int> c;
CachedStore<MemoryStore, int> d;
```

Template template parameters let us get the syntax we want. Note that you need to use the class keyword for template parameters that themselves have template parameters.

```
template <template <typename> class Store, typename T>
struct CachedStore2 {
  Store<T> store;
  Cache<T> cache;
};
```

```
CachedStore2<NetworkStore, int> e;
CachedStore2<MemoryStore, int> f;
```

Function try blocks
Function try blocks exist to catch errors thrown while evaluating a constructor's initializer list. You can't wrap a normal try-catch block around the initializer list because it exists outside the function body. To fix this, C++ allows a try-catch block to serve as the body of a function:

```
int f() { throw 0; }

// Here there is no way to catch the error thrown by f()
struct A {
  int a;
  A::A() : a(f()) {}
};

// The value thrown from f() can be caught if a try-catch block is used as
// the function body and the initializer list is moved after the try keyword
struct B {
  int b;
  B::B() try : b(f()) {
  } catch(int e) {
  }
};
```

Oddly enough, this syntax isn't just limited to constructors but is available for all function definitions.

Scope rules in C++ grammer
A class name may be hidden by the name of an object, function, or enu-merator declared in the same scope. If a class and an object, function, or enumerator is declared in the same scope (in any order) with the same name the class name is hidden. (Stroustrup, 1991)

```
missing_header.h
int T(int);
extern int a;
```

```
main.cpp
#include <missing_header.h>
class T { /* rest of code */ };

int main (){
T(a); /* parser would incorrectly interpret this
         as a declaration when header file is missing. This is
         in fact a function call! */
}
```

Even if the symbol table appears to be complete, symbols in a missing header file could have hidden symbols in the current compilation unit.