The underlying programming rules of C++ result in ambiguities. In this report, we discuss the following cases of ambiguity with examples of source code and resolution to the ambiguity:

- Dangling else
- Multiple inheritance
- Function overloading
- Most vexing parse
- Template vs greater than
- Destructor call vs one's complement

Reference: Meta-compilation for C++

1.Dangling else

This type of ambiguity arises in nested if-else when the target of the else clause is not explicitly defined. This problem can be solved by defining the scope of if-else statements using curly braces{}.

Ambiguous	Unambiguous

`if (condition1)`
`  if (condition2)`
`  doSomething();`
`else`
`  doSomethingElse();`
	`if (condition1) {`
	`  if (condition2) {`
	`  doSomething();`
	`  }`
	`}`
	`else {`
	`  doSomethingElse();`
	`}`

Example:

```
#include<stdio.h>

int main(int argc, char const *argv[])
{
    int a = 1;
    if( a )
        if( a < 1) printf("a < 1");

    else
    {
        printf("a = 0");
    }
```

```
    return 0;
}
```

Which outputs : a=0

While compiling with clang gives the following warning:

```
warning: add explicit braces to avoid dangling else [-Wdangling-else]
```

Unambiguous Code:

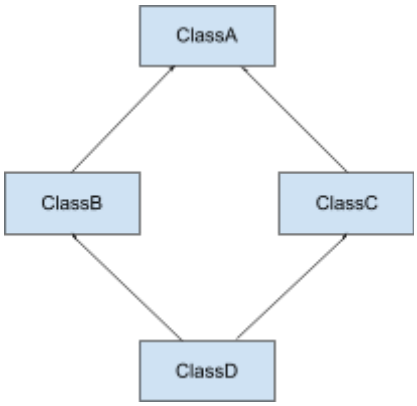
```
#include<stdio.h>

int main(int argc, char const *argv[])
{
    int a = 1;
    if( a ){
        if( a < 1) printf("a < 1");
    }
    else
    {
        printf("a = 0");
    }

    return 0;
}
```

2.Multiple inheritance

In C++, a class can inherit from multiple base classes. Hence, multiple inheritance may result in accessing the same member, causing ambiguity. The following image and program illustrate the diamond inheritance problem.




```
class ClassA
{
public:
    int a;
    void foo(){
        a=0;
    }
};
```

```
class ClassB :  public ClassA
{
public:
    int b;
};
class ClassC :  public ClassA
{
public:
    int c;
};
class ClassD : public ClassB, public ClassC
{
public:
    int d;
};
int main(int argc, char const *argv[])
{
    ClassD obj;
    obj.a = 1;
    obj.foo();
    return 0;
}
```

Clang compiler shows the following error:

```
27:9: error: non-static member 'a' found in multiple base-class
subobjects of type 'ClassA':
    class ClassD -> class ClassB -> class ClassA
    class ClassD -> class ClassC -> class ClassA
    obj.a = 1;
        ^
4:9: note: member found by ambiguous name lookup
    int a;
        ^
28:9: error: non-static member 'foo' found in multiple base-class
subobjects of type 'ClassA':
    class ClassD -> class ClassB -> class ClassA
    class ClassD -> class ClassC -> class ClassA
    obj.foo();
        ^
```

```
5:10: note: member found by ambiguous name lookup
    void foo(){
         ^
2 errors generated.
```

Inheritance related ambiguities can be resolved by virtual function or explicitly defining scope.

Virtual inheritance resolution:

```
class ClassB : virtual public ClassA
{
public:
    int b;
};
class ClassC : virtual public ClassA
{
public:
    int c;
};
```

Scope resolution:

```
int main(int argc, char const *argv[])
{
    ClassD obj;
    obj.ClassB::a = 1;
    obj.ClassC::foo();
    return 0;
}
```

Even the virtual derived classes can not access the base class members with the same name. So, if a derived class has multiple base classes with a same-named member, calling that member will result in ambiguity. In this case, the only resolution is to explicitly define scope or name members differently.

3. Function overloading

Function overloading is a feature of C++ where multiple functions with the same name can be defined in the same scope, given that each function has different parameter(s). The compiler determines which function to call depending on the argument type. However, this may create ambiguous situations where the compiler is unable to select the relevant function. Examples:

a. Argument related ambiguity:

In the following example, the compiler converts 10.2 to double, hence the ambiguity.

```
#include <stdio.h>
void foo(int i)
{ printf("%d\n", i);}

void foo(float f)
{ printf("%f\n",f );}

int main()
{
      foo(10);
      foo(10.2);
      return 0;
}
```

g++/g++ std c++11 output:

```
In function 'int main()':
11:10: error: call of overloaded 'foo(double)' is ambiguous
  foo(10.2);
         ^
2:6: note: candidate: void foo(int)
 void foo(int i)
      ^
5:6: note: candidate: void foo(float)
 void foo(float f)
      ^
```

clang output:

```
11:2: error: call to 'foo' is ambiguous
        foo(10.2);
        ^~~
2:6: note: candidate function
void foo(int i)
     ^
5:6: note: candidate function
void foo(float f)
     ^
1 error generated.
```

Resolution:

```
#include <stdio.h>
void foo(int i)
```

```
{ printf("%d\n", i);}

void foo(float f)
{ printf("%f\n",f );}

int main()
{
    foo(10);
    foo((float)10.2);
    return 0;
}
```

```
#include <stdio.h>
void foo(int i)
{ printf("%d\n", i);}

void foo(float f)
{ printf("%f\n",f );}

int main()
{
    foo(10);
    foo((float)10.2);
    return 0;
}
```

```
#include <stdio.h>
void foo(int i)
{ printf("%d\n", i);}

void foo(float f)
{ printf("%f\n",f );}

int main()
{
    foo(10);
```

```
    foo(10.2f);
    return 0;
}
```

Ambiguity due to default argument:

```
#include <stdio.h>

void foo(int i)
{ printf("%d\n",i );}

void foo(int i, int j=1)
{ printf("%d,%d\n",i,j );}

int main(){
    foo(10);
    return 0;
}
```

g++/g++ std c++11 output:

```
In function 'int main()':
10:8: error: call of overloaded 'foo(int)' is ambiguous
   10 |     foo(10);
      |     ~~~^~~~
3:6: note: candidate: 'void foo(int)'
    3 | void foo(int i)
      |      ^~~
6:6: note: candidate: 'void foo(int, int)'
    6 | void foo(int i, int j=1)
      |      ^~~
```

This can be resolved by removing default arguments or opting out of function overloading.

b. Reference parameters:

C++ functions can take value or reference as argument which may cause ambiguity:

```
#include <stdio.h>

void foo(int i){
    printf("value %d\n", i);
}
```

```
void foo(int& i){
    printf("lval %d\n", i);
}

int main(int argc, char const *argv[])
{
    int a = 1;
    foo(1);//calls foo(int i)
    foo(a);
    return 0;
}
```

g++/g++ std c++11 output:

```
In function 'int main(int, const char**)':
14:8: error: call of overloaded 'foo(int&)' is ambiguous
   14 |     foo(a);
      |     ~~~^~~
3:6: note: candidate: 'void foo(int)'
    3 | void foo(int i){
      |      ^~~
6:6: note: candidate: 'void foo(int&)'
    6 | void foo(int& i){
      |      ^~~
```

Resolution:

```
#include <stdio.h>
#include <iostream>
void foo(int i){
    printf("value %d\n", i);
}
void foo(int& i){
    printf("lval %d\n", i);
}

int main(int argc, char const *argv[])
{
    int a = 1;
    foo(1);
    foo(std::move(a));
    return 0;
```

```
}
```

Output:

```
value 1
value 1
```

Or pass by rvalue reference:

```
#include <stdio.h>

void foo(int &&i){
    printf("rvalue ref %d\n", i);
}
void foo(int& i){
    printf("lvalue %d\n", i);
}

int main(int argc, char const *argv[])
{
    int a = 1;
    foo(a);
    foo(2);
    return 0;
}
```

Output:

```
lvalue 1
rvalue ref 2
```

Reference:https://oneraynyday.github.io/dev/2017/09/04/C++-Lvalue-Rvalue-Move/

c.Library function

Sometimes, ambiguity may arise due to defining a function with the same name as a library function:

```
#include <iostream>
#include <vector>

using namespace std;

void plus (vector <int> *vec1, vector <int> *vec2)
{
```

```
    //code
}


int main()
{
    unsigned int i;
    vector <int> vec1, vec2;
    for (i = 1; i < 5; ++i)
    {
        vec1.push_back (i);
    }
    for (i = 9; i > 5; --i)
    {
        vec2.push_back (i);
    }
    plus (&vec1, &vec2);
    return 0;
}
```

Here, including the <iostream> header file results in introducing a function named plus in the same scope, which causes the ambiguity. g++ output:

```
In function 'int main()':
24:5: error: reference to 'plus' is ambiguous
   24 |     plus (&vec1, &vec2);
      |     ^~~~
In file included from /usr/include/c++/11/string:48,
                 from /usr/include/c++/11/bits/locale_classes.h:40,
                 from /usr/include/c++/11/bits/ios_base.h:41,
                 from /usr/include/c++/11/ios:42,
                 from /usr/include/c++/11/ostream:38,
                 from /usr/include/c++/11/iostream:39,
                 from 1:
/usr/include/c++/11/bits/stl_function.h:160:12: note: candidates are:
'template<class _Tp> struct std::plus'
  160 |     struct plus;
      |            ^~~~
6:6: note:                 'void plus(std::vector<int>*,
std::vector<int>*)'
```

```
    6 | void plus (vector <int> *vec1, vector <int> *vec2)
      |      ^~~~
```

Resolution:

Using scope accordingly.

```
#include <iostream>
#include <vector>

using namespace std;

void plus (vector <int> *n1, vector <int> *n2)
{
    //code
}


int main()
{
    unsigned int i;
    vector<int> n1, n2;
    for (i = 1; i < 5; ++i)
    {
        n1.push_back (i);
    }
    for (i = 9; i > 5; --i)
    {
        n2.push_back (i);
    }
    ::plus (&n1, &n2);
    return 0;
}
```

Or

```
#include <iostream>
#include <vector>

void plus (std::vector <int> *n1, std::vector <int> *n2)
{
    //code
```

```
}

int main()
{
    unsigned int i;
    std::vector<int> n1, n2;
    for (i = 1; i < 5; ++i)
    {
        n1.push_back (i);
    }
    for (i = 9; i > 5; --i)
    {
        n2.push_back (i);
    }
    plus (&n1, &n2);
    return 0;
}
```

4. Most vexing parse

The "most vexing parse" is a term coined by Scott Meyers for an ambiguity in C++ declaration syntax that leads to counterintuitive behavior. The following example:

```
#include <stdio.h>

struct A {
    explicit A() {
        printf("A constructor\n");
    }
};

struct B {
    explicit B(A a) {
        printf("B constructor\n");
    }
};

int main() {
    B b(A());
    return 0;
```

```
}
```

The line `B b(A());` is ambiguous because it can be interpreted as:

1. A variable b of struct B, initialized with an anonymous instance of struct A
2. A function declaration with the name b, returning an object of type B, and taking a single (unnamed) argument, which is a function returning an object of type A and taking no input

The C++ standard requires the second interpretation in both cases, even though the first interpretation is the intuitive one. g++ output of the above code:

```
In function 'int main()':
16:8: warning: parentheses were disambiguated as a function
declaration [-Wvexing-parse]
   16 |     B b(A());
      |        ^~~~
16:8: note: replace parentheses with braces to declare a variable
   16 |     B b(A());
      |        ^~~~
      |        -
      |        {   -
      |            }
```

Resolution:

1. C-style type conversion:

```
B b((A()));
```

2. Uniform initialization

```
B b(A{});
B b{A()};
B b{A{}};
```

5. Template vs greater than

In C++, template arguments are passed inside an angle bracket pair (<params>). However, the compiler considers the next > after the first < as the template ending. If > is intended to be used as a greater than operator inside the template argument section, the > will not perform the desired arithmetic operation but rather end the template argument section.

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

template <typename T, int value> class A {
```

```
public:
    T t;
    int val=value;
    A() {
        cout<<"constructor "<<val<<endl;
    }
};

int main(int argc, char const *argv[])
{
    A<int, 4>7 > a;
    return 0;
}
```

g++ output of the program:

```
In function 'int main(int, const char**)':
15:14: error: expected unqualified-id before numeric constant
   15 |     A<int, 4>7 > a;
      |              ^
```

This can be resolved by parenthesizing the greater than operation:

```
A<int, (4>7) > a;
```

6. Destructor call vs one's complement

In C++, an unqualified destructor name can not appear in an expression because of ambiguity with the complement operator. Even if C is a class, ~C() is treated as a unary complement rather than a destructor call.

```
#include <stdio.h>

class a{
public:
    a(){printf("%s\n", "Constructor");}
    ~a(){printf("%s\n", "Destructor");}
    void destructor(){
        ~a();
    }
};

int main(int argc, char const *argv[])
```

```
{
    a A;
    A.destructor();
    return 0;
}
```

The expression ~a() results in ambiguity. g++ output:

```
In member function 'void a::destructor()':
8:9: error: no match for 'operator~' (operand type is 'a')
    8 |         ~a();
      |         ^~~~
```

clang output:

```
8:9: error: invalid argument type 'a' to unary expression
        ~a();
        ^~~~
1 error generated.
```

Resolution:

```
#include <stdio.h>

class a{
public:
    a(){printf("%s\n", "Constructor");};
    ~a(){printf("%s\n", "Destructor");};
    void destructor(){
        this->~a();
    }
};

int main(int argc, char const *argv[])
{
    a A;
    A.destructor();
    return 0;
}
```