The Bug Type Taxonomy of OOP-Related Bugs in C++ Compilers

***Abstract*—This document details the taxonomy of OOP-related bug types in C++ compilers, which is a supplementary file of our paper submitted to ICSE-25, i.e., *A Comprehensive Study of OOP-Related Bugs in C++ Compilers*. We describe 10 principal categories of bug types and introduce the subcategories within each. Overall, the document covers 36 subcategories of OOP-related bugs in C++ compilers.**

I. Bug Types

A. Bug Type Identification Results

We first investigate the bug reports to categorize their bug types. We identified 10 main categories of bug types, each further divided into several subcategories.

① **Encapsulation Related Bugs.** This category of bugs is triggered by encapsulation-related language features of C++. C++ defines access control for derived classes or other classes, and C++ allows the `friend` keyword to access private or protect members of other classes. Moreover, the namespace and `using` statement also affect encapsulation. This complex mechanism usually leads to compiler bugs in implementing access control to class members. For this bug type, the most common root causes are missing invocation of access control APIs and incorrect branch conditions in lookup logic. It has 5 subcategories and we only introduce the following one.

1) Class member lookup bugs. These bugs occur due to the compiler failing to look up class members that are visible from the outer scope. Listing 1 shows one such bug, GCC-103081 [1], which cannot lookup the member `OINK` due to the wrong branch condition.

2) Member pointer accessing bugs. These bugs are triggered by the compiler incorrectly parsing the member pointers (including function pointers and data pointers) during assignments and conversions.

3) Class members access control bugs. These bugs occur due to the compiler failing to reference class members within the accessible scope or incorrectly introducing class members beyond legal scopes. In other words, the compiler can look up the member but incorrectly access it.

4) Friend function handling bugs. Friend functions enable two or more classes to share their `private` and `protected` members, which provides more flexibility to the access control in C++. These bugs are triggered by friend classes or friend functions.

5) Lambda capturing bugs. In C++, lambda expressions can access variables from the outer scope by their name, i.e., capturing the name. These bugs are triggered by the capturing feature.

```
/* ---- The bug-triggering program ---- */
  enum class Pig { OINK };
  struct Hog {
    using enum Pig;
    Hog(Pig) { }
  };
  template <unsigned> void pen() {
    (void) Hog(Hog::OINK);
  }
  void pen() {   pen<0>(); }
/* ---- The corresponding patch ---- */
-   if (DECL_NAMESPACE_SCOPE_P (t))
+   if (!uses_template_parms (DECL_CONTEXT (t)))
```

Listing 1: Class Member Lookup Bug GCC-103081

```
/* ---- The bug-triggering program ---- */
  struct A {};
  struct B : virtual A {};
  struct C : virtual A {};
  struct D : B,C { using A::A; };
```

Listing 2: Virtual Inheritances Handling Bug GCC-81164

② **Inheritance Related Bugs.** This category of bugs is triggered by complicated inheritance relationships or special corner inheritance cases. The compilers are error-prone regarding inheritance because C++ supports multiple inheritance, which could not only involve complex inheritance hierarchy but also introduce the concept of virtual base class. The most common root causes of this bug type are incorrect assignment of the flag values and overlooking corner cases. It has 4 subcategories and we introduce the following one.

1) Inherited class member related bugs. The bugs occur when the compiler fails to check for name conflicts between members of a subclass. Listing **??** shows one such example.

2) Empty base classes handling bugs. Empty base classes require the compiler to implicitly generate code or use some special process. These bugs are due to the compiler missing these special processes.

3) Virtual inheritances handling bugs. In C++, virtual inheritance is introduced to resolve member name conflicts caused by diamond inheritance. These bugs are caused by mishandling virtual inheritance, such as mixing members from base classes. Listing 2 shows one such bug in GCC [2], whose patch adds code blocks of the missing process.

4) Incorrect invocation of constructors/destructors. In C++, constructors/destructors in an inheritance hierarchy are selected and invoked in a specific order to ensure correct initialization and cleanup of object members. This type of bug occurs when the compiler incorrectly selects or orders these

```
/* ---- The bug-triggering program ---- */
  class a {
    virtual long b() const;
  };
  class c : a {
  public:
    long b() const;
  };
  class d : c {
    long e();
  };
  long d::e() { b(); }
/* ---- The corresponding patch ---- */
-   if (!fn || DECL_HAS_IN_CHARGE_PARM_P (fn))
+   if (!fn || DECL_HAS_VTT_PARM_P (fn))
```

Listing 3: Override-related Bug GCC-98744

functions between subclasses and base classes.

③ **Runtime Polymorphism Related Bugs.** This category of bugs is triggered by runtime polymorphism. Runtime polymorphism is one core feature of OOP in C++, which enables the program to decide the function to be executed during execution rather than at compile time. To achieve this, C++ compilers maintain virtual function tables for the base class and insert function pointers when the derived class overrides the base. The most common root cause is API misuse. It has 2 subcategories and we introduce the following one.

1) Override-related bugs. These bugs occur when the compiler implements polymorphism via override, especially wrongly building the virtual function table. Listing 3 shows one such bug [3], due to using the incorrect API.

2) Incorrect conversions from base to its derived classes. To safely cast a base class type to a derived class type, C++ requires runtime type information check, i.e., `dynamic_cast`. These bugs occur when the compiler fails to drive the type information for the conversion from a subclass to its base class.

④ **Compile-Time Polymorphism Related Bugs.** C++ achieves compile-time polymorphism via templates, which can be stamped out different instantiations under different type parameters. Templates introduce numerous syntactic rules and compile-time checks, often leading the compiler to overlook the handling of some cases. This category of bugs is triggered by parsing, instantiating, or specializing template classes. The most common root causes are missing corner cases and incorrect branch conditions. It has 5 subcategories and we introduce 2 of them.

1) Template parameter list parsing bugs. To trigger the bugs, the program should use the variadic template class, i.e., `<typename...>`. These bugs are caused by incorrect unpacking type parameter lists of template classes. Listing 4 shows one such example [4] causing the compiler crash, and its root cause is the misuse of APIs.

2) Static class member parsing bugs. These bugs are introduced by the conflict that the static member variables of a class need to be bound in the early stages of compilation, whereas template parameters can only be determined at the time of declaration of template class objects.

3) Dependent name handling bugs. These bugs are due to incorrectly determining the validity of a member access of

```
/* ---- The bug-triggering program ---- */
  template<typename> struct s {
     template<typename... Args> requires true
     static void f(Args...) { }
  };
  auto foo = s<int>::f();
```

Listing 4: Template Parameter List Parsing Bug LLVM-44658

```
/* ---- The bug-triggering program ---- */
  template <typename> struct S {
    struct A;
    struct f A();
  };
  template class S <int>;
/* ---- The corresponding patch ---- */
-   if (tree ti = (TREE_CODE (fn) == TYPE_DECL
-                  && !TYPE_DECL_ALIAS_P (fn)
-              ? TYPE_TEMPLATE_INFO (TREE_TYPE (fn))
-              : DECL_TEMPLATE_INFO (fn)))
+   if (tree ti = get_template_info (fn))
```

Listing 5: Template Parameter Instantiation Bug GCC-90916

a type parameter class during instantiating a template. For example, in a template class, whether `T::a` is legal depends on whether the type `T` passed in contains the member `a`.

4) Type deduction bugs. This type of bug occurs when the compiler, during the expansion of template classes, overlooks special cases in the type deduction process.

5) Template parameter instantiation bugs. This bug type is triggered by the type parameter instantiation of templates. Listing 5 shows one such example [5], which triggers the compiler crash on the valid code during instantiation, and the bug is caused by the incorrect branch condition. Note that the bug-triggering program is confusing, the code `struct f A()` is a function declaration with a returning type `struct f`, and the compiler should distinguish the function name `A` the data member `struct A`.

⑤ **Object Initialization Bugs.** This category of bugs is triggered by object initialization, i.e., the first stage during the lifetime of an object. The life cycle plays an important role in the OOP features of C++, where C++ objects contain many implicit or explicit convention rules during the initialization process, such as the choice of constructors and the initialization order of parent and child classes, which easily lead to compiler bugs. The most common root causes are API misuse and incorrect branch conditions. It has 5 subcategories and we introduce 2 of them.

1) List/direct initialization handling bugs. List initialization and direct initialization are special initialization approaches. This category of bugs is caused by the compiler incorrectly handling list/direct initialization. Listing 6 shows an invalid code that is accepted by LLVM [6]. The key to triggering this bug is the list initialization at the last line (i.e., `C c{...}`), and the root cause is the compiler missing this case.

2) Default constructor/destructor generation bugs. C++ compilers automatically generate default constructors and destructors for classes. This type of bug occurs when the compiler incorrectly generates them.

```
/* ---- The bug-triggering program ---- */
  struct A { int a; };
  struct B { int b; };
  struct C { A a;  B b; };
  C c{.a=0, .b={12}};
```

Listing 6: List Initialization Bug LLVM-49020

```
/* ---- The bug-triggering program ---- */
  struct A {
    constexpr A() { c[0] = 0; }
    char c[2];
  };
  constexpr A a;
```

Listing 7: Non-static Data Member Initializer Bug GCC-99700

3) Non-static data member initializer bugs. These bugs are caused by incorrectly handling complex expressions assigned to class members during object initialization. Listing 7 shows one such example [7], which is invalid but accepted by GCC, triggered by the initializer of `c[0]`.

4) Object array initialization bugs. This type of bug occurs due to the compiler incorrectly initializing an array whose elements are object types.

5) Interface class initialization bugs. In C++, a class could contain virtual functions. This category of bug occurs due to the compiler incorrectly initializing the object with virtual functions.

⑥ **Overloading Related Bugs.** Besides templates, C++ supports overloading, i.e., another form of compile-time polymorphism that supports multiple functions with the same name but different parameter lists. This category of bug is triggered by overloading, and has 3 subcategories.

1) Function overloading bugs. The bugs are caused by the compiler selecting the wrong overloaded member functions or constructors.

2) Operator overloading bugs. The bugs are triggered by operator overloading of classes.

3) Function name mangling bugs. To support overloading, C++ compilers automatically modify (i.e., mangle) the names of the overloaded functions. This category of bug is triggered by mangling.

⑦ **AST Visiting Bugs.** AST visiting relates to parsing and IR code generation. This category of bug is triggered by parsing complex expressions related to OOP-related operations, which has 3 subcategories.

1) Type conversion management bugs. The bugs occur when the compiler incorrectly processes explicit/implicit type conversions, including overriding the type cast operators and unconditional type casting.

2) R-value processing bugs. The bugs occur when the compiler incorrectly judges an L-value as an R-value during AST building.

3) Constant propagation/folding bugs in IR generation. The C++ compiler frond-end also performs early-stage optimization on AST, including constant propagation/folding. This category of bugs occurs when the compiler incorrectly implements front-end constant propagation/folding.

⑧ **Error Handling Bugs.** This category of bugs is caused by incorrect handling compilation errors, which has 3 subcategories.

1) Missing or wrong compilation error messages. The bugs occur due to missing warning/error messages or using inaccurate warning/error descriptions.

2) Useless compilation error messages. The bugs occur due to outputting unnecessary warning messages.

3) Error mark assignment bugs. The compilers use error marks when encountering compilation errors during parsing, that is, the compiler sets the mark and then unwinds the call stack until a proper handler is found. The bugs are caused by incorrectly setting error marks to wrong values or missing handlers.

⑨ **C++ New Feature Related Bugs.** This category of bug is triggered by new features of new C++ standards, such as C++17 and C++20, which has 5 subcategories.

1) Coroutines related bugs. Coroutines are introduced in C++20 that enable suspend and resume functions. The bugs occur due to compiler error handling coroutines in class member functions.

2) Structured binding bugs. Structure binding is a feature introduced in C++17, which unpacks the elements of structures, arrays, or types into separate variables. The bugs occur when the compiler fails to deduct types during binding.

3) Constraints and concepts related bugs. Constraints and concepts are features introduced in C++20 that enhance the expressiveness and safety of template programming. The bugs occur when the compiler fails to parse expressions of the new features.

4) Incorrect evaluation of `constexpr`/`consteval`. Constant expression (i.e., `constexpr`) is introduced in C++11, which indicates the expression should be constant at compile time. Constant evaluation (i.e., `consteval`) is introduced in C++20 that specifies that a function must be evaluated at compile time. This category of bugs occurs due to the compiler failing or incorrectly evaluating the constant.

5) Incorrect implementation of `noexcept`. The keyword `noexcept` is introduced in C++11, which is used to indicate whether a function would throw exceptions. The bugs occur when the compiler incorrectly implements the keyword.

⑩ **Others.** This category of bugs collects unusual and unique bugs that can not be classified into any other categories.

REFERENCES

[1] GCC-103081. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=103081

[2] GCC-81164. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=81164

[3] GCC-98744. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=98744

[4] LLVM-44658. Accessed: March 15, 2024. [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=44658

[5] GCC-90916. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90916

[6] LLVM-49020. Accessed: March 15, 2024. [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=49020

[7] GCC-99700. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99700