Implementation of visitor pattern in C++ by using std::variant

Ruslan Popov, Nadiia Karpenko
Faculty of Physics, Electronics and Computer Systems,
Oles Honchar Dnipro National University, Haharina Ave, 72, Dnipro

Abstract

In this paper we presented a new way to implement the Visitor pattern in C++ by using the standard container std::variant. We have shown how this method solves the expression problem and simplifies the creation of compilers and interpreters. We compared the classical approach with ours and highlighted all the pros and cons of the new solution.

When developing compilers and interpreters, the problem of organizing and modeling the nodes of an abstract syntax tree (AST) arises. These programs can perform various actions on the AST. For example, a compiler can optimize a program (the result of such an action is a new tree), an interpreter can execute a program, for example, a function can return a value, but it can also cause side effects, such as changing the state of an object, outputting information to the console or log file, etc. So, the problem arises of encoding the tree in such a way that you can easily add functions to process this tree. This problem is called "expression problem" in foreign literature [1].

Usually, in object-oriented programming languages such as C++ or Java, AST is represented as a hierarchy of classes. Methods of working on this tree are transferred to special classes that implement the `Visitor` interface. `Visitor` is a pattern that allows you to add new functions to classes without changing them. It turns out that the data of the AST node and the code that works on it are divided into two classes: the AST node and the visitor [3].

To implement the Visitor pattern in C++, first, an abstract `Visitor` class is created in which virtual methods for working on each node of the AST (`visitInteger`, `visitBinOp`, etc.) are described. After that, certain functionality is wrapped in a class that implements this interface. Each AST node must have a virtual `accept` method to accept a reference to an instance of the Visitor class and call the visit method in it, which corresponds to this AST node. This way, all the functionality of working on the AST is distributed to specific classes, so there is no need to change the definitions of nodes and you can easily add new methods.

The disadvantage of this approach is that when adding a new node of the AST, you need to perform a whole set of actions:

- Create a virtual visit method that corresponds to the new node;
- Implement the accept method in the node;
- Check the correctness of using types before implementing them;

- Add new functionality.

The question arises: can this pattern be reproduced more easily? The answer to this is given by the C++ standard, which describes a modern type-safe implementation of the union type (`std::variant`). In this case, access to union members is carried out by calling the `std::visit` function, which accepts a `Callable` object that must contain functions for working with the corresponding data type stored in `std::variant` [2].

The proposed way to implement the Visitor pattern in C++ is to represent abstract AST nodes (such as `Declaration`, `Statement`, `Expression`) using `std::variant`. Certain functionality over AST nodes is implemented as a separate class that contains functions that accept arguments of all AST nodes.

The advantages of this method are the use of the built-in capabilities of the C++ language, no need to create a separate `Visitor` interface, no need to change the AST nodes and create an `accept` method. Creating a new node consists in creating a separate class and adding it to `std::variant`. Since calling the necessary functions to work with the stored data is implemented automatically using the type system, the programmer is protected from spelling errors that may occur when writing `visit` methods.

But the biggest advantage of the proposed approach is that the methods that work on the AST can return any type of data. According to the classical implementation of the Visitor pattern, all `visit` functions are virtual, and therefore no template can be created over them. The optimizer can't return a new tree, the interpreter can't return the result of calculations, so they have to store all the results in the field of their class.

Our method has several drawbacks that you should pay attention to before using it:

1. `std::variant` is available only from C++17;
2. `std::variant` is implemented using template metaprogramming, so the compiler, in case of any errors in the code, can generate strange and complex error messages;
3. `Statement` or `Expression` types cease to be abstract polymorphic classes, instead they become unions, so the program may use more memory than necessary (but in fact, the Rust and CPython compilers implement AST in this way);
4. you should be careful when creating AST nodes, they cannot be used inside the `std::variant` node to which it belongs, as the classic problem of using a structure inside itself will arise.

Thus, in this article, we have compared the classic way of reproducing a AST in C++ using the Visitor pattern and proposed a new way of coding this pattern, which is faster to write and provides more opportunities.

[1] Expression problem, 2013. Link: https://wiki.c2.com/?ExpressionProblem.

[2] std::variant - cppreference.com, 2023. Link: https://en.cppreference.com/w/cpp/utility/variant.

[3] Visitor in c++ / design patterns, 2023. Link: https://refactoring.guru/design-patterns/visitor/cpp/example.