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

Popov Ruslan (first-year student, group KI-23-1), Karpenko Nadiia

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

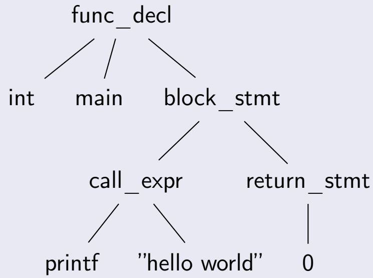
MEICS, November 2023

Abstract syntax tree

The source code:

```
int main()
{
    printf("hello world");
    return 0;
}
```

Parsing result (simplified):



func_decl

int

main

block_stmt

call_expr

return_stmt

printf

"hello world"

0

Functionality over AST

print: $AST \to String$

print $Var(a) = "a"$

print $UnaryOp(-, Var(b)) = " - b"$

eval: $AST \to Value$

eval $Integer(7) = 7$

eval $BinOp(Integer(2), +, Integer(2)) = 4$

optimize: $AST \to AST$

optimize $BinOp(Var(a), *, 4) = BinOp(Var(a), <<, 2)$

First attempt to solve expression problem

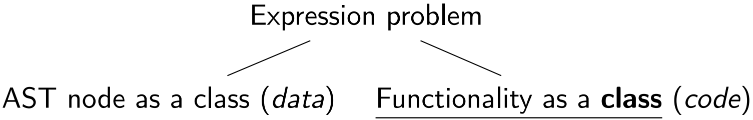
Node interface

```
struct Node {
    virtual ~Node() {}
    virtual std::string ToString() = 0;
    virtual int Eval() = 0;
    virtual Node* Optimize() = 0;
};
```

Binary expression node

```
struct BinOp : Node {
    Expr* left, right;
    BinOpType op;
    ...
    virtual std::string ToString();
    virtual int Eval();
    virtual Node* Optimize();
};
```

Classical approach: Visitor pattern



Expression problem

AST node as a class (*data*)

Functionality as a **class** (*code*)

An AST node

```
struct BinOp : Node {
    Expr* left, right;
    BinOpType op;

    virtual void Accept(Visitor* vis) {
        vis->VisitBinOp(this);^^I
    }
};
```

Visitor

```
struct Visitor {
    virtual void VisitInteger(Integer* expr) = 0;
    virtual void VisitUnaryOp(UnaryOp* expr) = 0;
    virtual void VisitBinOp(BinOp* expr) = 0;
}
```

Problems with Visitor pattern

- Not so easy to understand.
- A lot of things to do in order to add a new AST node:
  1. Create a new method `Visit` to `Visitor` interface.
  2. Create a new method `Accept` in the new AST node.

A question

Can we implement this pattern easier?

std::variant in C++

Until C++17

```cpp
// A simplified example.
struct SumType {
    int AsInteger();
    bool IsInteger();

    ValueType GetType();

private:
    ValueType type;
    union {
        int num;
        ...
    } as;
};
```

Since C++17

```cpp
#include <variant>

// Greatly reduced code size.
using SumType =
    std::variant<int, ...>;
```

std::visit y C++

Declaration of std::visit since C++17

```
template <class Visitor, class... Variants>
constexpr /* ... */ visit(Visitor&& vis, Variants&&... vars);
```

Usage example:

```
struct MyVisitor {
    std::string operator()(int arg)   { return "integer"; }
    std::string operator()(bool arg)  { return "boolean"; }
};

int main() {
    std::variant<int, bool> var = 10;
    MyVisitor vis;
    std::cout << std::visit(vis, var) << std::endl;
    return 0;
}
```

Our approach to expression problem

Represent abstract AST nodes as variants

```
// A greately simplified example.
using Expr = std::variant<Int, Unary, Binary, ...>;
using Stmt = std::variant<If, While, Return, ...>;
```

```
struct Printer {
    std::string operator()(Int& expr);
    ...
    std::string operator()(If& stmt);
    ...
    std::string PrintExpr(Expr* expr) {
        return std::visit(*this, *expr);
    }
};
```

Some problems with our approach

- `std::variant` is available since C++17 → earlier standards are not supported.
- `std::variant` uses template metaprogramming → complex error reports.
- Do not use an `Expr` inside an `Expr` or a `Stmt` inside a `Stmt` directly → structure inside itself.

End of the report

Conclusions

End of the report

Thanks for attention!

Additional information

Source code: https://github.com/InAnYan/AstVisitor.
Email: popov_ro@fffeks.dnu.edu.ua.

Authors

Popov Ruslan (first-year student, group KI-23-1), Karpenko Nadiia.
Faculty of Physics, Electronics and Computer Systems, Oles Honchar Dnipro National University, Haharina Ave, 72, Dnipro