1 Definition of StmtVisitor:

Below is the definition of the class StmtVisitor:

template<typename ImplClass, typename RetTy=void>
class StmtVisitor: public StmtVisitorBase<make_ptr, ImplClass, RetTy> {};

The StmtVisitor is an implementation of class template StmtVisitorBase. The declaration of StmtVisitorBase is as below:

template<template <typename> class Ptr, typename ImplClass, typename RetTy=void>
class StmtVisitorBase{…}

As we can see from the above, the Ptr in StmtVisitor is the template make_ptr, defined as below:

template <typename T> struct make_ptr { typedef T *type; };

2 Macro Definitions in the StmtVisitorBase:

a) ***#define PTR(CLASS) typename Ptr<CLASS>::type***

PTR(CLASS) refers to a pointer to type 'CLASS', so assume PTR(CLASS) as a pointer

b) ***#define DISPATCH(NAME, CLASS)*** \
return static_cast<ImplClass*>(this)->Visit ## NAME(static_cast<PTR(CLASS)>(S))

Visit##NAME is to concate Visit and Name to a single string.

3 Visit()

3.1 Structure:

- The structure of Visit is as below:

```
RetTy Visit(PTR(Stmt) S)
{
    if (PTR(BinaryOperator) BinOp = dyn_cast<BinaryOperator>(S))
{Block 1}
    else if (PTR(UnaryOperator) UnOp = dyn_cast<UnaryOperator>(S))
{Block 2}
    switch (S->getStmtClass())
        {Block 3}
}
```

- Explanations:

a) PTR(Stmt) S: As we know, Ptr=make_ptr when applied to StmtVisitor, PTR(Stmt)=typename make_ptr<Stmt>::type=typename (Stmt*), which means S=(Stmt *); **S is of type 'Stmt*'** . The same with PTR(BinaryOperator) and PTR(UnaryOperator);

b) If S is a Binary Operation Expression such as “a+b”, Block 1 will call corresponding visit functions to visit S; If S is an Unary Operation Expression such as “a++”, Block 2 will call corresponding visit functions to visit S;

3.2 Block 1:

3.2.1 Code

```
if (PTR(BinaryOperator) BinOp = dyn_cast<BinaryOperator>(S))
        {
          switch (BinOp->getOpcode()) {
          case BO_PtrMemD:   DISPATCH(BinPtrMemD,   BinaryOperator);
          case BO_PtrMemI:   DISPATCH(BinPtrMemI,   BinaryOperator);
          case BO_Mul:       DISPATCH(BinMul,       BinaryOperator);
          case BO_Div:       DISPATCH(BinDiv,       BinaryOperator);
          case BO_Rem:       DISPATCH(BinRem,       BinaryOperator);
          case BO_Add:       DISPATCH(BinAdd,       BinaryOperator);
          case BO_Sub:       DISPATCH(BinSub,       BinaryOperator);
          case BO_Shl:       DISPATCH(BinShl,       BinaryOperator);
          case BO_Shr:       DISPATCH(BinShr,       BinaryOperator);

          case BO_LT:        DISPATCH(BinLT,        BinaryOperator);
          case BO_GT:        DISPATCH(BinGT,        BinaryOperator);
          case BO_LE:        DISPATCH(BinLE,        BinaryOperator);
          case BO_GE:        DISPATCH(BinGE,        BinaryOperator);
          case BO_EQ:        DISPATCH(BinEQ,        BinaryOperator);
          case BO_NE:        DISPATCH(BinNE,        BinaryOperator);

          case BO_And:       DISPATCH(BinAnd,       BinaryOperator);
          case BO_Xor:       DISPATCH(BinXor,       BinaryOperator);
          case BO_Or :       DISPATCH(BinOr,        BinaryOperator);
          case BO_LAnd:      DISPATCH(BinLAnd,      BinaryOperator);
          case BO_LOr :      DISPATCH(BinLOr,       BinaryOperator);
          case BO_Assign:    DISPATCH(BinAssign,    BinaryOperator);
          case BO_MulAssign: DISPATCH(BinMulAssign, CompoundAssignOperator);
          case BO_DivAssign: DISPATCH(BinDivAssign, CompoundAssignOperator);
          case BO_RemAssign: DISPATCH(BinRemAssign, CompoundAssignOperator);
          case BO_AddAssign: DISPATCH(BinAddAssign, CompoundAssignOperator);
          case BO_SubAssign: DISPATCH(BinSubAssign, CompoundAssignOperator);
          case BO_ShlAssign: DISPATCH(BinShlAssign, CompoundAssignOperator);
          case BO_ShrAssign: DISPATCH(BinShrAssign, CompoundAssignOperator);
          case BO_AndAssign: DISPATCH(BinAndAssign, CompoundAssignOperator);
          case BO_OrAssign:  DISPATCH(BinOrAssign,  CompoundAssignOperator);
          case BO_XorAssign: DISPATCH(BinXorAssign, CompoundAssignOperator);
          case BO_Comma:     DISPATCH(BinComma,     BinaryOperator);
            }
        }
```

3.2.2 Analysis:

We can choose a single line of the switch to analysis, all the others are similar:

case BO_Add: DISPATCH(BinAdd, BinaryOperator);

Below is the macro definition of DISPATCH:

#define DISPATCH(NAME, CLASS) \

return static_cast<ImplClass*>(this)->Visit ## NAME(static_cast<PTR(CLASS)>(S))

Thus, ***DISPATCH(BinAdd, BinaryOperator)(BinAdd corresponds to NAME, and BinaryOperator corresponds to CLASS)*** will be replaced by statement below(Assume the ImplClass is ***ScalarExprEmitter***):

return static_cast<ScalarExprEmitter *>(this)->VisitBinAdd(static_cast<BinaryOperator *>(S))

The above means: if S is a Addition Expression, it will call the function of VisitBinAdd of the ImplClass (in our case, ScalarExprEmitter) to visit S.(Functions such as VisitBinAdd is implemented in ScalarExprEmitter)

3.3 Block 2:

Block 2 is the same as Block 1, except it is invoked when S is an Unary Operation Expression.

3.4 Block 3:

```
switch (S->getStmtClass()) {
    default: llvm_unreachable("Unknown stmt kind!");
#define ABSTRACT_STMT(STMT)
#define STMT(CLASS, PARENT)                                   \
    case Stmt::CLASS ## Class: DISPATCH(CLASS, CLASS);
#include "clang/AST/StmtNodes.inc"
}
```

3.4.1 StmtNodes.inc(is modified in Clang-IOC):

Parts of the content of the file:

```
#ifndef EXPR
#   define EXPR(Type, Base) STMT(Type, Base)
#endif
ABSTRACT_STMT(EXPR(Expr, Stmt))

#ifndef CASTEXPR
#   define CASTEXPR(Type, Base) EXPR(Type, Base)
#endif
ABSTRACT_STMT(CASTEXPR(CastExpr, Expr))
#ifndef EXPLICITCASTEXPR
#   define EXPLICITCASTEXPR(Type, Base) CASTEXPR(Type, Base)
#endif
ABSTRACT_STMT(EXPLICITCASTEXPR(ExplicitCastExpr, CastExpr))
#ifndef CSTYLECASTEXPR
#   define CSTYLECASTEXPR(Type, Base) EXPLICITCASTEXPR(Type, Base)
#endif
CSTYLECASTEXPR(CStyleCastExpr, ExplicitCastExpr)
#undef CSTYLECASTEXPR
```

#ifndef CXXFUNCTIONALCASTEXPR

 # define CXXFUNCTIONALCASTEXPR(Type, Base) EXPLICITCASTEXPR(Type, Base)

#endif

CXXFUNCTIONALCASTEXPR(CXXFunctionalCastExpr, ExplicitCastExpr)

#undef CXXFUNCTIONALCASTEXPR

#ifndef CXXNAMEDCASTEXPR

 # define CXXNAMEDCASTEXPR(Type, Base) EXPLICITCASTEXPR(Type, Base)

#endif

ABSTRACT_STMT(CXXNAMEDCASTEXPR(CXXNamedCastExpr, ExplicitCastExpr))

#ifndef CXXNAMEDCASTEXPR

 # define CXXNAMEDCASTEXPR(Type, Base) EXPLICITCASTEXPR(Type, Base)

#endif

ABSTRACT_STMT(CXXNAMEDCASTEXPR(CXXNamedCastExpr, ExplicitCastExpr))

…

…

#ifndef OBJCBRIDGEDCASTEXPR

 # define OBJCBRIDGEDCASTEXPR(Type, Base) EXPLICITCASTEXPR(Type, Base)

#endif

OBJCBRIDGEDCASTEXPR(ObjCBridgedCastExpr, ExplicitCastExpr)

#undef OBJCBRIDGEDCASTEXPR

STMT_RANGE(ExplicitCastExpr, CStyleCastExpr, ObjCBridgedCastExpr)

#undef EXPLICITCASTEXPR

3.4.2 Analysis of how ExplicitCastExpr invoked:

In the file of "clang/AST/StmtNodes.inc", there is such a statement(shown above):

CSTYLECASTEXPR(CStyleCastExpr, ExplicitCastExpr)

The following codes in StmtNodes.inc told us to replace *CSTYLECASTEXPR(CStyleCastExpr, ExplicitCastExpr)* with *EXPLICITCASTEXPR (CStyleCastExpr, ExplicitCastExpr)*

#ifndef CSTYLECASTEXPR

 # define CSTYLECASTEXPR(Type, Base) EXPLICITCASTEXPR(Type, Base)

#endif

Then, the following codes in StmtNodes.inc told us to replace *EXPLICITCASTEXPR (CStyleCastExpr, ExplicitCastExpr)* with *CASTEXPR(CStyleCastExpr, ExplicitCastExpr):*

#ifndef EXPLICITCASTEXPR

 # define EXPLICITCASTEXPR(Type, Base) CASTEXPR(Type, Base)

#endif

Then, the following codes in StmtNodes.inc told us to replace *CASTEXPR(CStyleCastExpr, ExplicitCastExpr)*with *EXPR(CStyleCastExpr, ExplicitCastExpr):*

#ifndef CASTEXPR

 # define CASTEXPR(Type, Base) EXPR(Type, Base)

#endif

Then, the following codes in StmtNodes.inc told us to replace ***EXPR(CStyleCastExpr, ExplicitCastExpr)*** with ***STMT(CStyleCastExpr, ExplicitCastExpr)***

```
#ifndef EXPR
#   define EXPR(Type, Base) STMT(Type, Base)
#endif
```

Then , in StmtVisitor.h:

```
#define STMT(CLASS, PARENT)                                    \
      case Stmt::CLASS ## Class: DISPATCH(CLASS, CLASS);
```

However, when DISPATCH is expanded, ***STMT(CStyleCastExpr, ExplicitCastExpr)*** corresponds to

```
case StmtCStyleCastExprClass:
(static_cast<ImplClass*>(this))->VisitCStyleCastExpr (static_cast<CStyleCastExpr *>(S))
```

In fact, the if VisitCStyleCastExpr is not defined in the ImplClass, it will call the VisitCStyleCastExpr defined in StmtVisit.h, it is defined with macro. Details below.

4 Dealing with Visit Functions not defined in ImplClass:

Followed the definition of Visit(), there are codes below:

```
#define STMT(CLASS, PARENT)                                    \
   RetTy Visit ## CLASS(PTR(CLASS) S) { DISPATCH(PARENT, PARENT); }
#include "clang/AST/StmtNodes.inc"
```

As we can see, after expanding StmtNodes.inc, the original ***STMT(CStyleCastExpr, ExplicitCastExpr)*** (mentioned above) will be as follows(when ImplClass is Emit ScalarExprEmitter):

```
RetTy VisitCStyleCastExpr(CStyleCastExpr * S)
{
Return static_cast<ScalarExprEmitter*>(this)->VisitExplicitCastExpr(static_cast<ExplicitCastExpr *>(S))
}
```

The above means, if there is VisitCStyleCastExpr in the ImplClass(For example: ScalarExprEmitter), it will call that in the ImplClass if S belongs to StmtCStyleCastExprClass(in the switch block of Visit()), and if it is not defined in ImplClass, then it will call that defined in this base class, defined above, and this will clall the VisitExplicitCastExpr defined in the ImplClass.