编译原理研讨课实验PR001实验报告

任务说明

基于PRJ1函数标注的基础上，实现pragma ElementWise标注的语义内容：即实现静态数组的“+”、“=”、“*”操作，并将之反映在AST上，以备PR3产生中间代码。

成员组成

陈灿宇
宋鹏皓
金越

实验设计

设计思路

> 1.AST设计：
>
> > (1) 左右值问题：

本实验一个关键的问题在于，静态数组名（比如 `int C [100]`的C）是一个不可修改的(Unmodifiable)的左值(Lvalue)，因此它如果单独出现在等号左边（而不是`C[0]`这种对元素的赋值），那么编译器就会报错（`error:"expression not assignable"`）。这里采用的思路是在调用`CheckForModifiable`函数（这个函数在这里调用的作用是检查等号的左表达式是否是modifiable Lvalue）时，如果满足`#pragma ElementWise`标注，并且左表达式是个静态数组的数组名，则返回modifiable Lvalue的结果（这个函数中返回false）。

> > (2) 表达式返回值类型：

检查表达式结果的类型的工作，是由一系列`CheckXXXXOperands()`函数完成的，这些函数根据表达式的左表达式和右表达式（AST上反映为OP结点的左子树和右子树），以及其他一些必要信息返回该表达式结果的类型。我们需要做的就是更改`+`、`*`、`=`对应的Check函数，在满足`#pragma ElementWise`标注的情况下，返回表达式左表达式的类型（即静态数组）。

> 2.操作数匹配
>
> > (1) 静态数组检查

我们将表达式左表达式和右表达式都是静态数组类型作为最外围的判断条件（当然前提是满足`#pragma ElementWise`标注，后面不再赘述）（如课堂ppt上那样，左表达式右表达式都是`ConstanArrayType`），否则将不会进行ElementWise的处理。比如一个静态数组+一个指针这种表达式就不进行`ElementWise`处理，那么`clang`编译器就会和正常一样报错。这样做的好处是如"指针 = 静态数组"或者"静态数组 = 指针"这种表达式不进行ElementWise处理，而让编译器自然判断（后者会报错）。

> > (2) 大小检查

我们将表达式左表达式和右表达式静态数组的元素个数相等作为ElementWise处理的第一个筛选条件，若不满足，则不进行后续的处理，并报错：`error :"array range is incompatible"`。

> > (3) 类型相同

我们将表达式左表达式和右表达式静态数组的类型相同作为ElementWise处理的第二个筛选条件，若不满足，则不进行后续的处理，并报错：`error :"array element type is incompatible"`。
并且我们限定，进行ElementWise操作的只能是int类型的静态数组，这时第三个筛选条件，不满足则会报错：`error :"on the right(或left) of the operand, array element type is not integer"`

> 3.赋值类型不匹配

在`=`的Check函数中，会检查赋值的类型是否匹配，比如将指针赋值给静态数组就会报错（`error:"assigng to 'int [1000]' from incompatible type 'pointer'"`）.即使前面的modifiable Lvalue更改后这个问题还是不能解决，因为两者是在不同的函数里判断的，modifiable Lvalue是在`CheckForModifiableLvalue()`里判定的，而类型匹配则由`CheckSingleAssignmentConstraints()`判定。那这里的处理和modifiable Lvalue一样，满足条件直接返回Compatible.

> 4.有无`#pragma elementWise`标注的区分

在`PRJ1`中我们已经在`Sema`类里添加了一个`elementWise`标记，幸运的是`PRJ2`的大部分函数都在`Sema`类里，于是我们可以将这个`elementWise`标记作为一切有关ElementWise操作的前提。

实验实现

> 1. `llvm-3.3\tools\clang\lib\Sema\SemaExpr.cpp`中
>
> > 1) `CheckForModifiableLvalue()`函数,添加代码如下。该函数在`Sema::CheckAssignmentOperands()`中被调用，这里的更改对应于设计中将静态数组改为modifiable Lvalue。这里不能直接用"左表达式为静态数组"这种简单的条件判定，因为左边的静态数组可能是一个`const`量，像这样的就连lvalue都不是。这里的判定方法是获取表达式的`Classification`类信息，这是一个`Expr`类的嵌套类，它有两个字段`Kind`和

> `Modifiable`，而可赋值的静态数组（非`const`）的`Kind`是`CL_LValue`，说明它首先是个左值。`Modifiable`是`CM_ArrayType`即数组类型。

```
static bool CheckForModifiableLvalue(Expr *E, SourceLocation Loc, Sema &S) {
  assert(!E->hasPlaceholderType(BuiltinType::PseudoObject));

  if(S.IsElementWise == 1) {
    Expr::Classification VC = E->ClassifyModifiable(S.Context, Loc);
    if(VC.getKind() == Expr::Classification::CL_LValue &&
       VC.getModifiable() == Expr::Classification::CM_ArrayType) {
      return false;
    }
  }
...
}
```

> 2) `CheckAdditionOperands()`函数，添加代码如下。该函数对应`+`操作符。对应于设计中的操作数匹配，包括大小、类型以及静态数组限定。第一层if是`ElementWise`标注以及左右表达式都是静态数组的判定。第二层if是数组大小相同的判定，对应的else将错误信息存入Diag，将会打印出`error: "array range is incompatible"`。第三层if是元素类型的判定，相加的两者的元素类型相同并且都只能为`int`类型，对应的else将错误信息存入Diag,将会打印出`error: "array element type is incompatible"`。第四层if将数组类型限定在`int`数组。最里层将左右表达式中非右值的表达式转换为右值。正确的话最终返回左表达式的类型作为加法表达式的结果类型。

```
QualType Sema::CheckAdditionOperands( // C99 6.5.6
    ExprResult &LHS, ExprResult &RHS, SourceLocation Loc, unsigned Opc,
    QualType* CompLHSTy) {
  checkArithmeticNull(*this, LHS, RHS, Loc, /*isCompare=*/false);

  if (LHS.get()->getType()->isVectorType() ||
      RHS.get()->getType()->isVectorType()) {
    QualType compType = CheckVectorOperands(LHS, RHS, Loc, CompLHSTy);
    if (CompLHSTy) *CompLHSTy = compType;
    return compType;
  }

  if(IsElementWise == 1 &&
     ConstantArrayType::classof(LHS.get()->getType().getTypePtr()) &&
     ConstantArrayType::classof(RHS.get()->getType().getTypePtr())){
    const Expr* lhs_expr = LHS.get();
    const Type* lhs_type = lhs_expr -> getType().getTypePtr();
    const Expr* rhs_expr = RHS.get();
    const Type* rhs_type = rhs_expr -> getType().getTypePtr();

    const ConstantArrayType* lhs_t = dyn_cast<ConstantArrayType>(lhs_type);
    const ConstantArrayType* rhs_t = dyn_cast<ConstantArrayType>(rhs_type);

    const llvm::APInt lhs_data_num = lhs_t->getSize();
    const llvm::APInt rhs_data_num = rhs_t->getSize();

    QualType lhs_data_t = lhs_t -> getElementType().getUnqualifiedType();
    QualType rhs_data_t = rhs_t -> getElementType().getUnqualifiedType();

    if(lhs_data_num == rhs_data_num){
      if(lhs_data_t == rhs_data_t){
        if(lhs_data_t->isOnlyIntegerType() && rhs_data_t->isOnlyIntegerType()){
          if(!(lhs_expr -> isRValue())){
            Qualifiers tmp;
            ImplicitCastExpr *l_to_r_lhs = ImplicitCastExpr::Create(Context, \
              Context.getUnqualifiedArrayType(lhs_expr -> getType().getUnqualifiedType(), tmp), \
              CK_LValueToRValue, const_cast<Expr*>(lhs_expr), 0, VK_RValue);
            LHS = l_to_r_lhs;
          }
          if(!(rhs_expr -> isRValue())){
            Qualifiers tmp;
            ImplicitCastExpr *l_to_r_rhs = ImplicitCastExpr::Create(Context,
              Context.getUnqualifiedArrayType(rhs_expr -> getType().getUnqualifiedType(), tmp), \
              CK_LValueToRValue, const_cast<Expr*>(rhs_expr), 0, VK_RValue);
            RHS = l_to_r_rhs;
          }
          return LHS.get() -> getType();
        }
        else if(!lhs_data_t->isOnlyIntegerType()) {
          this->Diag(Loc, diag::err_left_array_element_type_not_integer);
          return QualType();
        }

        else if(!rhs_data_t->isOnlyIntegerType()) {
```

```
          this->Diag(Loc, diag::err_right_array_element_type_not_integer);
          return QualType();
        }
      }
      else {
        this->Diag(Loc, diag::err_array_element_type_incompatible);
        return QualType();
      }
    }
    else {
      this->Diag(Loc, diag::err_array_range_incompatible);
      return QualType();
    }
  }
...
}
```

> 3) `CheckMultiplyDivideOperands()`函数，对应`*`操作符，和加法修改一样，这里不再赘述。

> 4) `CheckAssignmentOperands()`函数，对应`=`操作符，与其他两个操作符修改相似，只不过没有改变成右值。

```
QualType Sema::CheckAssignmentOperands(Expr *LHSExpr, ExprResult &RHS,
                                       SourceLocation Loc,
                                       QualType CompoundType) {
  assert(!LHSExpr->hasPlaceholderType(BuiltinType::PseudoObject));

  if(IsElementWise == 1 &&
     ConstantArrayType::classof(LHSExpr->getType().getTypePtr()) &&
     ConstantArrayType::classof(RHS.get()->getType().getTypePtr())){
    const Expr* lhs_expr = LHSExpr;
    const Type* lhs_type = lhs_expr -> getType().getTypePtr();
    const Expr* rhs_expr = RHS.get();
    const Type* rhs_type = rhs_expr -> getType().getTypePtr();

    const ConstantArrayType* lhs_t = dyn_cast<ConstantArrayType>(lhs_type);
    const ConstantArrayType* rhs_t = dyn_cast<ConstantArrayType>(rhs_type);

    const llvm::APInt lhs_data_num = lhs_t->getSize();
    const llvm::APInt rhs_data_num = rhs_t->getSize();

    QualType lhs_data_t = lhs_t -> getElementType().getUnqualifiedType();
    QualType rhs_data_t = rhs_t -> getElementType().getUnqualifiedType();

    if(lhs_data_num != rhs_data_num){
      this->Diag(Loc, diag::err_array_range_incompatible);
      return QualType();
    }
    if(lhs_data_t != rhs_data_t) {
      this->Diag(Loc, diag::err_array_element_type_incompatible);
      return QualType();
    }
    if(!lhs_data_t->isOnlyIntegerType()) {
      this->Diag(Loc, diag::err_left_array_element_type_not_integer);
      return QualType();
    }
    if(!rhs_data_t->isOnlyIntegerType()) {
      this->Diag(Loc, diag::err_right_array_element_type_not_integer);
      return QualType();
    }
  }
...
}
```

> 5) `CheckSingleAssignmentConstraints()`函数,添加代码：该函数在`Sema::CheckAssignmentOperands()`中被调用（检查赋值语句的类型匹配），对应于设计中所说的将右表达式为静态数组的表达式类型直接判定为`Compatible`。这里的判读条件可以用赋值语句的右部是一个静态数组来判定。

```
Sema::CheckSingleAssignmentConstraints(QualType LHSType, ExprResult &RHS,
                                       bool Diagnose) {

  if(IsElementWise == 1
    && ConstantArrayType::classof(RHS.get()->getType().getTypePtr())) {
    return Compatible;
  }

 ....
}
```

2. `llvm-3.3\tools\clang\include\clang\AST\Type.h` 中，对Type添加判定int类型的函数，用于表达式左右部类型的判定。

```
class Type : public ExtQualsTypeCommonBase {
...
public:
  bool isOnlyIntegerType() const;
...
}

inline bool Type::isOnlyIntegerType() const {
  if (const BuiltinType *BT = dyn_cast<BuiltinType>(CanonicalType))
    return BT->getKind() == BuiltinType::Int;
  return false;
}
```

3. `llvm-3.3\tools\clang\include\clang\Basic\DiagnosticSemaKinds.td`中添加打印的错误信息：

```
def err_array_range_incompatible : Error<
  "array range is incompatible">;
def err_array_element_type_incompatible : Error<
  "array element type is incompatible">;
def err_left_array_element_type_not_integer : Error<
  "on the left of the operand, array element type is not integer">;
def err_right_array_element_type_not_integer : Error<
  "on the right of the operand, array element type is not integer">;
```

其它

总结

实验结果总结

分成员总结

> 组员：金越
> 这次实验完成了#pragma elementWise 编译制导语义的构建AST部分，相较于PRJ1有一定难度的提升，一是因为没有老师的实现说明文档，需要自己查找很多信息。在改`CheckMultiplyDivideOperands()`时，深入跟踪了很多的函数去寻找制止静态数组赋值报错的地方，最后根据错误信息定位到

> diag::err_typecheck_array_not_modifiable_lvalue，才知道判断Modifiable Lvalue的条件。实验中也遇到很多坑，比如类型匹配时，一开始以为是`CheckAssignmentConstraints()`函数，但没看到它前面还调用了`CheckSingleAssignmentConstraints()`函数，导致出现了莫名奇妙的现象：明明改了函数，但还是会报类型不匹配的错误。其实前者是判断符合赋值,如'+='赋值号的类型匹配。实验对C++的要求也很高，包括嵌套类的调用（在使用Classification时）。不过好在这些在clang里都能找到相似的例子。关于左右值问题，老师上课用的方法是将静态数组转换为指针来使静态数组成为一个Modifiable Lvalue，但我们采用的时直接更改它在`CheckMultiplyDivideOperands()`中的判定条件，实际上这种方法比较简单粗暴，有潜在的BUG，但在目前的test看来是没问题的。总的来说，PRJ2的确很有挑战性，和其他项目不同的是，挑战性来源于对clang代码的熟悉程度和理解深度。

> 组员：陈灿宇
> 本次实验的难点在于入手比较困难，在代码量巨大的情况下，想要分析清楚每一行代码的作用是比较困难的，必须要有针对性地进行分析，结合BUG来进行分析。在实验过程中我们遇到的第一个比较难以解决的BUG是“error: array type 'int [1000]' is not assignable”，我们结合了多种思路来解决这个BUG，首先利用GDB跟踪来小范围定位BUG并且梳理出现BUG时某些关键变量的值，然后理清函数的调用关系，对于关键部分进行分析，同时还结合注释掉某些语句之后来分析某些语句的作用。最后的方法还是比较简单粗暴的，直接将整形数组类型在有pragma elementWise的情况下修改为Modifiable Lvalue。后面还遇到了一个比较关键的BUG是“error: assigng to 'int [1000]' from incompatible type 'pointer'"，但解决思路与第一个BUG比较类似。另外比较有趣的一点是，由于本次实验老师规定了只能对于integer类型数组进行运算，其它类型的数组必须要报错，这里必须要重新定义一个isOnlyIntegerType()函数对于integer进行识别，如果利用原有的isIntegerType()函数的话，它会在某些运算的时候将char类型转化成integer类型。

> 组员：宋鹏皓
> 在老师上课讲解后，本次实验的实验思路还算比较清晰，只是在具体操作上由于对框架和C++的不熟悉，我们还是碰到了一些难题。比如如果不修改assign函数，那么对于A[]= B[] + C[] 的表达式是会编译报错的，原因是A[]属于数组，这在具体代码里将其判断为不是可修改的左值。具体而言这里的数组是左值，但是是不可修改的。改动思路也很简单，就是将其状态修改为modifiable.但难点在于如何找到合适的地方去修改。我们花费了很大的功夫去跟踪不同的函数和类，不断比较和判断每一次修改是否会影响其它地方。所以整体而言，本次实验的内容并不复杂，但考验的是深入理解代码，正确调用各个函数接口的能力，这部分就比较费神费时间了。我也很高兴自己这方面的能力得到了锻炼。

成员贡献

> 组员：金越 ： 共同讨论实现代码，撰写实验报告
> 组员：陈灿宇 ： 共同讨论实现代码，补充实验报告
> 组员：宋鹏皓 ： 共同讨论实现代码，补充实验报告

测试结果

> 执行命令：`sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_1.c (P2_testcase_1.c ~ P2_testcase_12.c)`

> 测试结果如下：

- 实验1 ：

  测试程序：P2*testcase*1.c

```
#pragma elementWise
void foo1(){
  int A[1000];
  int B[1000];
  int C[1000];
  C = A + B;
  C = A * B;
  C = A;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_1.c
TranslationUnitDecl 0x67feb60 <<invalid sloc>>
|-TypedefDecl 0x67ff040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x67ff0a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x67ff3f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x67ff490 <./PR002/test/P2_testcase_1.c:2:1, line:9:1> foo1 'void ()'
  `-CompoundStmt 0x682c180 <line:2:12, line:9:1>
    |-DeclStmt 0x67ff608 <line:3:3, col:14>
    | `-VarDecl 0x67ff5b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x67ff6b8 <line:4:3, col:14>
    | `-VarDecl 0x67ff660 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x67ff768 <line:5:3, col:14>
    | `-VarDecl 0x67ff710 <col:3, col:13> C 'int [1000]'
    |-BinaryOperator 0x67ff850 <line:6:3, col:11> 'int [1000]' '='
    | |-DeclRefExpr 0x67ff780 <col:3> 'int [1000]' lvalue Var 0x67ff710 'C' 'int [1000]'
    | `-BinaryOperator 0x67ff828 <col:7, col:11> 'int [1000]' '+'
    |   |-ImplicitCastExpr 0x67ff7f8 <col:7> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x67ff7a8 <col:7> 'int [1000]' lvalue Var 0x67ff5b0 'A' 'int [1000]'
    |   `-ImplicitCastExpr 0x67ff810 <col:11> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x67ff7d0 <col:11> 'int [1000]' lvalue Var 0x67ff660 'B' 'int [1000]'
    |-BinaryOperator 0x682c0e0 <line:7:3, col:11> 'int [1000]' '='
    | |-DeclRefExpr 0x682c010 <col:3> 'int [1000]' lvalue Var 0x67ff710 'C' 'int [1000]'
    | `-BinaryOperator 0x682c0b8 <col:7, col:11> 'int [1000]' '*'
    |   |-ImplicitCastExpr 0x682c088 <col:7> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x682c038 <col:7> 'int [1000]' lvalue Var 0x67ff5b0 'A' 'int [1000]'
    |   `-ImplicitCastExpr 0x682c0a0 <col:11> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x682c060 <col:11> 'int [1000]' lvalue Var 0x67ff660 'B' 'int [1000]'
    `-BinaryOperator 0x682c158 <line:8:3, col:7> 'int [1000]' '='
      |-DeclRefExpr 0x682c108 <col:3> 'int [1000]' lvalue Var 0x67ff710 'C' 'int [1000]'
      `-DeclRefExpr 0x682c130 <col:7> 'int [1000]' lvalue Var 0x67ff5b0 'A' 'int [1000]'
[testing] ./PR002/test/P2_testcase_1.c
```

实验证明，针对有pragma的加法，乘法，赋值的位运算能正确生成AST

- 实验2：

  测试程序：P2*testcase*2.c

```
void foo2(){
  int A[1000];
  int B[1000];
  int C[1000];
  C = A + B;
  C = A * B;
  C = A;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_2.c
./PR002/test/P2_testcase_2.c:5:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A + B;
      ~ ^ ~
./PR002/test/P2_testcase_2.c:6:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A * B;
      ~ ^ ~
./PR002/test/P2_testcase_2.c:7:5: error: array type 'int [1000]' is not assignable
  C = A;
  ~ ^
TranslationUnitDecl 0x6989b60 <<invalid sloc>>
|-TypedefDecl 0x698a040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x698a0a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x698a3f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x698a490 <./PR002/test/P2_testcase_2.c:1:1, line:8:1> foo2 'void ()'
  `-CompoundStmt 0x69b7418 <line:1:12, line:8:1>
    |-DeclStmt 0x698a608 <line:2:3, col:14>
    | `-VarDecl 0x698a5b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x698a6b8 <line:3:3, col:14>
    | `-VarDecl 0x698a660 <col:3, col:13> B 'int [1000]'
    `-DeclStmt 0x698a768 <line:4:3, col:14>
      `-VarDecl 0x698a710 <col:3, col:13> C 'int [1000]'
3 errors generated.
[testing] ./PR002/test/P2_testcase_2.c
```

实验证明，针对无pragma的加法，乘法，赋值的位运算不能正确生成AST，同时会报错

- 实验3：

  测试程序：P2*testcase*3.c

```
#pragma elementWise
void foo3(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  C = D;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_3.c
./PR002/test/P2_testcase_3.c:7:5: error: assigning to 'int [1000]' from incompatible type 'int *'
  C = D;
    ^ ~
TranslationUnitDecl 0x6bf2b60 <<invalid sloc>>
|-TypedefDecl 0x6bf3040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6bf30a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6bf33f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6bf3490 <./PR002/test/P2_testcase_3.c:2:1, line:8:1> foo3 'void ()'
  `-CompoundStmt 0x6c20370 <line:2:12, line:8:1>
    |-DeclStmt 0x6bf3608 <line:3:3, col:14>
    | `-VarDecl 0x6bf35b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x6bf36b8 <line:4:3, col:14>
    | `-VarDecl 0x6bf3660 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x6bf3768 <line:5:3, col:14>
    | `-VarDecl 0x6bf3710 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x6bf3818 <line:6:3, col:9>
      `-VarDecl 0x6bf37c0 <col:3, col:8> D 'int *'
1 error generated.
[testing] ./PR002/test/P2_testcase_3.c
```

实验证明，对于有pragma情况下指针赋值给数组会报错。

- 实验4：

测试程序：P2*testcase*4.c

```
#pragma elementWise
void foo4(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  (A + B) = C;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_4.c
./PR002/test/P2_testcase_4.c:7:11: error: expression is not assignable
  (A + B) = C;
  ~~~~~~~ ^
TranslationUnitDecl 0x51e2b60 <<invalid sloc>>
|-TypedefDecl 0x51e3040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x51e30a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x51e33f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x51e3490 <./PR002/test/P2_testcase_4.c:2:1, line:8:1> foo4 'void ()'
  `-CompoundStmt 0x52100a0 <line:2:12, line:8:1>
    |-DeclStmt 0x51e3608 <line:3:3, col:14>
    | `-VarDecl 0x51e35b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x51e36b8 <line:4:3, col:14>
    | `-VarDecl 0x51e3660 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x51e3768 <line:5:3, col:14>
    | `-VarDecl 0x51e3710 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x51e3818 <line:6:3, col:9>
      `-VarDecl 0x51e37c0 <col:3, col:8> D 'int *'
1 error generated.
[testing] ./PR002/test/P2_testcase_4.c
```

实验证明，(A + B) = C的情况会报错。

- 实验5：

  测试程序：P2*testcase*5.c

```
#pragma elementWise
void foo5(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  C = A + D;
  C = D + A;
  C = D + D;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_5.c
./PR002/test/P2_testcase_5.c:7:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A + D;
      ~ ^ ~
./PR002/test/P2_testcase_5.c:8:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = D + A;
      ~ ^ ~
./PR002/test/P2_testcase_5.c:9:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = D + D;
      ~ ^ ~
TranslationUnitDecl 0x699cb60 <<invalid sloc>>
|-TypedefDecl 0x699d040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x699d0a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x699d3f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x699d490 <./PR002/test/P2_testcase_5.c:2:1, line:10:1> foo5 'void ()'
  `-CompoundStmt 0x69ca520 <line:2:12, line:10:1>
    |-DeclStmt 0x699d608 <line:3:3, col:14>
    | `-VarDecl 0x699d5b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x699d6b8 <line:4:3, col:14>
    | `-VarDecl 0x699d660 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x699d768 <line:5:3, col:14>
    | `-VarDecl 0x699d710 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x699d818 <line:6:3, col:9>
      `-VarDecl 0x699d7c0 <col:3, col:8> D 'int *'
3 errors generated.
[testing] ./PR002/test/P2_testcase_5.c
```

实验证明，指针与数组的加法运算会报错。

- 实验6：

  测试程序：P2*testcase*6.c

```c
#pragma elementWise
void foo6(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  (A + B) = C;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_6.c
./PR002/test/P2_testcase_6.c:7:11: error: expression is not assignable
  (A + B) = C;
  ~~~~~~~ ^
TranslationUnitDecl 0x6e0bb60 <<invalid sloc>>
|-TypedefDecl 0x6e0c040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6e0c0a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6e0c3f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6e0c490 <./PR002/test/P2_testcase_6.c:2:1, line:8:1> foo6 'void ()'
  `-CompoundStmt 0x6e390a0 <line:2:12, line:8:1>
    |-DeclStmt 0x6e0c608 <line:3:3, col:14>
    | `-VarDecl 0x6e0c5b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x6e0c6b8 <line:4:3, col:14>
    | `-VarDecl 0x6e0c660 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x6e0c768 <line:5:3, col:14>
    | `-VarDecl 0x6e0c710 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x6e0c818 <line:6:3, col:9>
      `-VarDecl 0x6e0c7c0 <col:3, col:8> D 'int *'
1 error generated.
[testing] ./PR002/test/P2_testcase_6.c
```

实验证明，(A + B) = C的情况会报错。

- 实验7：

  测试程序：P2*testcase*7.c

```
#pragma elementWise
void foo7(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  int E[10][100];
  E = A;
  E = A + B;
  E = A * B;
}
```

  实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_7.c
./PR002/test/P2_testcase_7.c:8:5: error: array range is incompatible
  E = A;
    ^
./PR002/test/P2_testcase_7.c:9:5: error: array range is incompatible
  E = A + B;
    ^
./PR002/test/P2_testcase_7.c:10:5: error: array range is incompatible
  E = A * B;
    ^
TranslationUnitDecl 0x5e86b60 <<invalid sloc>>
|-TypedefDecl 0x5e87040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5e870a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5e873f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5e87490 <./PR002/test/P2_testcase_7.c:2:1, line:11:1> foo7 'void ()'
  `-CompoundStmt 0x5eb4340 <line:2:12, line:11:1>
    |-DeclStmt 0x5e87608 <line:3:3, col:14>
    | `-VarDecl 0x5e875b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x5e876b8 <line:4:3, col:14>
    | `-VarDecl 0x5e87660 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x5e87768 <line:5:3, col:14>
    | `-VarDecl 0x5e87710 <col:3, col:13> C 'int [1000]'
    |-DeclStmt 0x5e87818 <line:6:3, col:9>
    | `-VarDecl 0x5e877c0 <col:3, col:8> D 'int *'
    `-DeclStmt 0x5eb4138 <line:7:3, col:17>
      `-VarDecl 0x5eb40e0 <col:3, col:16> E 'int [10][100]'
3 errors generated.
[testing] ./PR002/test/P2_testcase_7.c
```

  实验证明，赋值给二位数组的位运算能正确生成AST。

- 实验8：

  测试程序：P2*testcase*8.c

```
#pragma elementWise
void foo8(){
  int A[1000];
  int B[1000];
  const int C[1000];
  C = A;
  C = A + B;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_8.c
./PR002/test/P2_testcase_8.c:6:5: error: read-only variable is not assignable
  C = A;
  ~ ^
./PR002/test/P2_testcase_8.c:7:5: error: read-only variable is not assignable
  C = A + B;
  ~ ^
TranslationUnitDecl 0x5174b60 <<invalid sloc>>
|-TypedefDecl 0x5175040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x51750a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x51753f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5175490 <./PR002/test/P2_testcase_8.c:2:1, line:8:1> foo8 'void ()'
  `-CompoundStmt 0x51a2068 <line:2:12, line:8:1>
    |-DeclStmt 0x5175608 <line:3:3, col:14>
    | `-VarDecl 0x51755b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x51756b8 <line:4:3, col:14>
    | `-VarDecl 0x5175660 <col:3, col:13> B 'int [1000]'
    `-DeclStmt 0x51757a8 <line:5:3, col:20>
      `-VarDecl 0x5175750 <col:3, col:19> C 'const int [1000]'
2 errors generated.
[testing] ./PR002/test/P2_testcase_8.c
```

实验证明，将int型数组赋值给const int型数组不能正确生成AST，同时会报错。

- 实验9：

  测试程序：P2*testcase*9.c

```c
#pragma elementWise
void foo9(){
  int A[1000];
  const int B[1000];
  int C[1000];
  C = B;
  C = A + B;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_9.c
TranslationUnitDecl 0x6e98b60 <<invalid sloc>>
|-TypedefDecl 0x6e99040 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6e990a0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6e993f0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6e99490 <./PR002/test/P2_testcase_9.c:2:1, line:8:1> foo9 'void ()'
  `-CompoundStmt 0x6ec60b8 <line:2:12, line:8:1>
    |-DeclStmt 0x6e99608 <line:3:3, col:14>
    | `-VarDecl 0x6e995b0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x6e996f8 <line:4:3, col:20>
    | `-VarDecl 0x6e996a0 <col:3, col:19> B 'const int [1000]'
    |-DeclStmt 0x6e997a8 <line:5:3, col:14>
    | `-VarDecl 0x6e99750 <col:3, col:13> C 'int [1000]'
    |-BinaryOperator 0x6e99810 <line:6:3, col:7> 'int [1000]' '='
    | |-DeclRefExpr 0x6e997c0 <col:3> 'int [1000]' lvalue Var 0x6e99750 'C' 'int [1000]'
    | `-DeclRefExpr 0x6e997e8 <col:7> 'const int [1000]' lvalue Var 0x6e996a0 'B' 'const int [1000]'
    `-BinaryOperator 0x6ec6090 <line:7:3, col:11> 'int [1000]' '='
      |-DeclRefExpr 0x6e99838 <col:3> 'int [1000]' lvalue Var 0x6e99750 'C' 'int [1000]'
      `-BinaryOperator 0x6ec6068 <col:7, col:11> 'int [1000]' '+'
        |-ImplicitCastExpr 0x6ec6038 <col:7> 'int [1000]' <LValueToRValue>
        | `-DeclRefExpr 0x6e99860 <col:7> 'int [1000]' lvalue Var 0x6e995b0 'A' 'int [1000]'
        `-ImplicitCastExpr 0x6ec6050 <col:11> 'int [1000]' <LValueToRValue>
          `-DeclRefExpr 0x6ec6010 <col:11> 'const int [1000]' lvalue Var 0x6e996a0 'B' 'const int [10
00]'
[testing] ./PR002/test/P2_testcase_9.c
```

实验证明，有const int型数组的加法，赋值（将const int型数组赋值给int型数组）运算能正确生成AST。

- 实验10 ：

  测试程序：P2*testcase*10.c

```
#pragma elementWise
void foo10(){
  int A[1000];
  int B[1000];
  int C[1000];
  int D[1000];
  D = A + B + C;
  D = A * B + C;
  D = (D = A + B);
  D = (A + B) * C;
  D = (A + B) * (C + D);
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_10.c
TranslationUnitDecl 0x5cd0b80 <<invalid sloc>>
|-TypedefDecl 0x5cd1060 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5cd10c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5cd1410 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5cd14b0 <./PR002/test/P2_testcase_10.c:2:1, line:12:1> foo10 'void ()'
  `-CompoundStmt 0x5cfe7e0 <line:2:13, line:12:1>
    |-DeclStmt 0x5cd1628 <line:3:3, col:14>
    | `-VarDecl 0x5cd15d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x5cd16d8 <line:4:3, col:14>
    | `-VarDecl 0x5cd1680 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x5cd1788 <line:5:3, col:14>
    | `-VarDecl 0x5cd1730 <col:3, col:13> C 'int [1000]'
    |-DeclStmt 0x5cd1838 <line:6:3, col:14>
    | `-VarDecl 0x5cd17e0 <col:3, col:13> D 'int [1000]'
    |-BinaryOperator 0x5cfe168 <line:7:3, col:15> 'int [1000]' '='
    | |-DeclRefExpr 0x5cd1850 <col:3> 'int [1000]' lvalue Var 0x5cd17e0 'D' 'int [1000]'
    | `-BinaryOperator 0x5cfe140 <col:7, col:15> 'int [1000]' '+'
    |   |-BinaryOperator 0x5cfe0d8 <col:7, col:11> 'int [1000]' '+'
    |   | |-ImplicitCastExpr 0x5cfe0a8 <col:7> 'int [1000]' <LValueToRValue>
    |   | | `-DeclRefExpr 0x5cd1878 <col:7> 'int [1000]' lvalue Var 0x5cd15d0 'A' 'int [1000]'
    |   | `-ImplicitCastExpr 0x5cfe0c0 <col:11> 'int [1000]' <LValueToRValue>
    |   |   `-DeclRefExpr 0x5cfe080 <col:11> 'int [1000]' lvalue Var 0x5cd1680 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x5cfe128 <col:15> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x5cfe100 <col:15> 'int [1000]' lvalue Var 0x5cd1730 'C' 'int [1000]'
    |-BinaryOperator 0x5cfe2c8 <line:8:3, col:15> 'int [1000]' '='
    | |-DeclRefExpr 0x5cfe190 <col:3> 'int [1000]' lvalue Var 0x5cd17e0 'D' 'int [1000]'
    | `-BinaryOperator 0x5cfe2a0 <col:7, col:15> 'int [1000]' '+'
    |   |-BinaryOperator 0x5cfe238 <col:7, col:11> 'int [1000]' '*'
    |   | |-ImplicitCastExpr 0x5cfe208 <col:7> 'int [1000]' <LValueToRValue>
    |   | | `-DeclRefExpr 0x5cfe1b8 <col:7> 'int [1000]' lvalue Var 0x5cd15d0 'A' 'int [1000]'
    |   | `-ImplicitCastExpr 0x5cfe220 <col:11> 'int [1000]' <LValueToRValue>
    |   |   `-DeclRefExpr 0x5cfe1e0 <col:11> 'int [1000]' lvalue Var 0x5cd1680 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x5cfe288 <col:15> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x5cfe260 <col:15> 'int [1000]' lvalue Var 0x5cd1730 'C' 'int [1000]'
    |-BinaryOperator 0x5cfe430 <line:9:3, col:17> 'int [1000]' '='
    | |-DeclRefExpr 0x5cfe2f0 <col:3> 'int [1000]' lvalue Var 0x5cd17e0 'D' 'int [1000]'
    | `-ParenExpr 0x5cfe410 <col:7, col:17> 'int [1000]'
    |   `-BinaryOperator 0x5cfe3e8 <col:8, col:16> 'int [1000]' '='
    |     |-DeclRefExpr 0x5cfe318 <col:8> 'int [1000]' lvalue Var 0x5cd17e0 'D' 'int [1000]'
    |     `-BinaryOperator 0x5cfe3c0 <col:12, col:16> 'int [1000]' '+'
    |       |-ImplicitCastExpr 0x5cfe390 <col:12> 'int [1000]' <LValueToRValue>
    |       | `-DeclRefExpr 0x5cfe340 <col:12> 'int [1000]' lvalue Var 0x5cd15d0 'A' 'int [1000]'
    |       `-ImplicitCastExpr 0x5cfe3a8 <col:16> 'int [1000]' <LValueToRValue>
    |         `-DeclRefExpr 0x5cfe368 <col:16> 'int [1000]' lvalue Var 0x5cd1680 'B' 'int [1000]'
    |-BinaryOperator 0x5cfe5b0 <line:10:3, col:17> 'int [1000]' '='
    | |-DeclRefExpr 0x5cfe458 <col:3> 'int [1000]' lvalue Var 0x5cd17e0 'D' 'int [1000]'
    | `-BinaryOperator 0x5cfe588 <col:7, col:17> 'int [1000]' '*'
    |   |-ParenExpr 0x5cfe528 <col:7, col:13> 'int [1000]'
    |   | `-BinaryOperator 0x5cfe500 <col:8, col:12> 'int [1000]' '+'
    |   |   |-ImplicitCastExpr 0x5cfe4d0 <col:8> 'int [1000]' <LValueToRValue>
    |   |   | `-DeclRefExpr 0x5cfe480 <col:8> 'int [1000]' lvalue Var 0x5cd15d0 'A' 'int [1000]'
    |   |   `-ImplicitCastExpr 0x5cfe4e8 <col:12> 'int [1000]' <LValueToRValue>
    |   |     `-DeclRefExpr 0x5cfe4a8 <col:12> 'int [1000]' lvalue Var 0x5cd1680 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x5cfe570 <col:17> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x5cfe548 <col:17> 'int [1000]' lvalue Var 0x5cd1730 'C' 'int [1000]'
    `-BinaryOperator 0x5cfe7b8 <line:11:3, col:23> 'int [1000]' '='
      |-DeclRefExpr 0x5cfe5d8 <col:3> 'int [1000]' lvalue Var 0x5cd17e0 'D' 'int [1000]'
      `-BinaryOperator 0x5cfe790 <col:7, col:23> 'int [1000]' '*'
        |-ParenExpr 0x5cfe6a8 <col:7, col:13> 'int [1000]'
        | `-BinaryOperator 0x5cfe680 <col:8, col:12> 'int [1000]' '+'
        |   |-ImplicitCastExpr 0x5cfe650 <col:8> 'int [1000]' <LValueToRValue>
        |   | `-DeclRefExpr 0x5cfe600 <col:8> 'int [1000]' lvalue Var 0x5cd15d0 'A' 'int [1000]'
        |   `-ImplicitCastExpr 0x5cfe668 <col:12> 'int [1000]' <LValueToRValue>
        |     `-DeclRefExpr 0x5cfe628 <col:12> 'int [1000]' lvalue Var 0x5cd1680 'B' 'int [1000]'
        `-ParenExpr 0x5cfe770 <col:17, col:23> 'int [1000]'
          `-BinaryOperator 0x5cfe748 <col:18, col:22> 'int [1000]' '+'
            |-ImplicitCastExpr 0x5cfe718 <col:18> 'int [1000]' <LValueToRValue>
            | `-DeclRefExpr 0x5cfe6c8 <col:18> 'int [1000]' lvalue Var 0x5cd1730 'C' 'int [1000]'
            `-ImplicitCastExpr 0x5cfe730 <col:22> 'int [1000]' <LValueToRValue>
              `-DeclRefExpr 0x5cfe6f0 <col:22> 'int [1000]' lvalue Var 0x5cd17e0 'D' 'int [1000]'
```

```
[testing] ./PR002/test/P2_testcase_10.c
```

实验证明，对带有括号的运算能正确生成AST。

- 实验11 ：

  测试程序：P2*testcase*11.c

```c
int main(){}

#pragma elementWise
void foo1(){
    int A[1000];
    int B[100];
    int C[1000];
    C = A + B;
    C = A * B;
}

#pragma elementWise
void foo2(){
    int A[1000];
    char B[1000];
    int C[1000];
    C = A + B;
    C = A * B;
}

#pragma elementWise
void foo3(){
    char A[1000];
    char B[1000];
    char C[1000];
    C = A + B;
    C = A * B;
    C = A;
}

#pragma elementWise
void foo4(){
    double A[1000];
    double B[1000];
    double C[1000];
    C = A + B;
    C = A * B;
    C = A;
    char   D[1000];
    C = D;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_11.c
./PR002/test/P2_testcase_11.c:8:11: error: array range is incompatible
    C = A + B;
          ^
./PR002/test/P2_testcase_11.c:9:11: error: array range is incompatible
    C = A * B;
          ^
./PR002/test/P2_testcase_11.c:17:11: error: array element type is incompatible
    C = A + B;
          ^
./PR002/test/P2_testcase_11.c:18:11: error: array element type is incompatible
    C = A * B;
          ^
./PR002/test/P2_testcase_11.c:26:11: error: on the left of the operand, array element type is not
      integer
    C = A + B;
          ^
./PR002/test/P2_testcase_11.c:27:11: error: on the left of the operand, array element type is not
      integer
    C = A * B;
          ^
./PR002/test/P2_testcase_11.c:28:7: error: on the left of the operand, array element type is not
      integer
    C = A;
        ^
./PR002/test/P2_testcase_11.c:36:11: error: on the left of the operand, array element type is not
      integer
    C = A + B;
          ^
./PR002/test/P2_testcase_11.c:37:11: error: on the left of the operand, array element type is not
      integer
    C = A * B;
          ^
./PR002/test/P2_testcase_11.c:38:7: error: on the left of the operand, array element type is not
      integer
    C = A;
        ^
./PR002/test/P2_testcase_11.c:40:7: error: array element type is incompatible
    C = D;
        ^
TranslationUnitDecl 0x7033b80 <<invalid sloc>>
|-TypedefDecl 0x7034060 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x70340c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x7034410 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
|-FunctionDecl 0x70344b0 <./PR002/test/P2_testcase_11.c:1:1, col:12> main 'int ()'
| `-CompoundStmt 0x7034550 <col:11, col:12>
|-FunctionDecl 0x70345c0 <line:4:1, line:10:1> foo1 'void ()'
| `-CompoundStmt 0x7061360 <line:4:12, line:10:1>
|   |-DeclStmt 0x7034738 <line:5:5, col:16>
|   | `-VarDecl 0x70346e0 <col:5, col:15> A 'int [1000]'
|   |-DeclStmt 0x7034828 <line:6:5, col:15>
|   | `-VarDecl 0x70347d0 <col:5, col:14> B 'int [100]'
|   `-DeclStmt 0x7061258 <line:7:5, col:16>
|     `-VarDecl 0x7061200 <col:5, col:15> C 'int [1000]'
|-FunctionDecl 0x70613b0 <line:13:1, line:19:1> foo2 'void ()'
| `-CompoundStmt 0x7061790 <line:13:12, line:19:1>
|   |-DeclStmt 0x70614e8 <line:14:5, col:16>
|   | `-VarDecl 0x7061490 <col:5, col:15> A 'int [1000]'
|   |-DeclStmt 0x70615d8 <line:15:5, col:17>
|   | `-VarDecl 0x7061580 <col:5, col:16> B 'char [1000]'
|   `-DeclStmt 0x7061688 <line:16:5, col:16>
|     `-VarDecl 0x7061630 <col:5, col:15> C 'int [1000]'
|-FunctionDecl 0x70617e0 <line:22:1, line:29:1> foo3 'void ()'
| `-CompoundStmt 0x7061bd0 <line:22:12, line:29:1>
|   |-DeclStmt 0x7061918 <line:23:5, col:17>
|   | `-VarDecl 0x70618c0 <col:5, col:16> A 'char [1000]'
|   |-DeclStmt 0x70619c8 <line:24:5, col:17>
|   | `-VarDecl 0x7061970 <col:5, col:16> B 'char [1000]'
|   `-DeclStmt 0x7061a78 <line:25:5, col:17>
|     `-VarDecl 0x7061a20 <col:5, col:16> C 'char [1000]'
`-FunctionDecl 0x7061c20 <line:32:1, line:41:1> foo4 'void ()'
```

```
-FunctionDecl 0x7061c20 <line:32:1, line:41:1> f004 'void ()
 `-CompoundStmt 0x7062150 <line:32:12, line:41:1>
   |-DeclStmt 0x7061d98 <line:33:5, col:19>
   | `-VarDecl 0x7061d40 <col:5, col:18> A 'double [1000]'
   |-DeclStmt 0x7061e48 <line:34:5, col:19>
   | `-VarDecl 0x7061df0 <col:5, col:18> B 'double [1000]'
   |-DeclStmt 0x7061ef8 <line:35:5, col:19>
   | `-VarDecl 0x7061ea0 <col:5, col:18> C 'double [1000]'
   `-DeclStmt 0x70620e8 <line:39:5, col:19>
     `-VarDecl 0x7062090 <col:5, col:18> D 'char [1000]'
11 errors generated.
[testing] ./PR002/test/P2_testcase_11.c
```

实验证明，对于（有pragma）类型不匹配，非int型数组，数组长度不匹配的情况不能生成AST，同时会报错。

- 实验12 ：

  测试程序：P2*testcase*12.c

```
int main(){}

//#pragma elementWise
void foo1(){
    int A[1000];
    int B[100];
    int C[1000];
    C = A + B;
    C = A * B;
}

//#pragma elementWise
void foo2(){
    int A[1000];
    char B[1000];
    int C[1000];
    C = A + B;
    C = A * B;
}

//#pragma elementWise
void foo3(){
    char A[1000];
    char B[1000];
    char C[1000];
    C = A + B;
    C = A * B;
    C = A;
}

//#pragma elementWise
void foo4(){
    double A[1000];
    double B[1000];
    double C[1000];
    C = A + B;
    C = A * B;
    C = A;
    char   D[1000];
    C = D;
}
```

实验结果：

```
[clang9@host2 ~]$ sh ./PR002/scripts/compile_and_check.sh ./PR002/test/P2_testcase_12.c
./PR002/test/P2_testcase_12.c:8:11: error: invalid operands to binary expression
      ('int *' and 'int *')
    C = A + B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:9:11: error: invalid operands to binary expression
      ('int *' and 'int *')
    C = A * B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:17:11: error: invalid operands to binary expression
      ('int *' and 'char *')
    C = A + B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:18:11: error: invalid operands to binary expression
      ('int *' and 'char *')
    C = A * B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:26:11: error: invalid operands to binary expression
      ('char *' and 'char *')
    C = A + B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:27:11: error: invalid operands to binary expression
      ('char *' and 'char *')
    C = A * B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:28:7: error: array type 'char [1000]' is not assignable
    C = A;
    ~ ^
./PR002/test/P2_testcase_12.c:36:11: error: invalid operands to binary expression
      ('double *' and 'double *')
    C = A + B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:37:11: error: invalid operands to binary expression
      ('double *' and 'double *')
    C = A * B;
        ~ ^ ~
./PR002/test/P2_testcase_12.c:38:7: error: array type 'double [1000]' is not assignable
    C = A;
    ~ ^
./PR002/test/P2_testcase_12.c:40:7: error: array type 'double [1000]' is not assignable
    C = D;
    ~ ^
TranslationUnitDecl 0x635fb80 <<invalid sloc>>
|-TypedefDecl 0x6360060 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x63600c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6360410 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
|-FunctionDecl 0x63604b0 <./PR002/test/P2_testcase_12.c:1:1, col:12> main 'int ()'
| `-CompoundStmt 0x6360550 <col:11, col:12>
|-FunctionDecl 0x63605c0 <line:4:1, line:10:1> foo1 'void ()'
| `-CompoundStmt 0x638d758 <line:4:12, line:10:1>
|   |-DeclStmt 0x6360738 <line:5:5, col:16>
|   | `-VarDecl 0x63606e0 <col:5, col:15> A 'int [1000]'
|   |-DeclStmt 0x6360828 <line:6:5, col:15>
|   | `-VarDecl 0x63607d0 <col:5, col:14> B 'int [100]'
|   `-DeclStmt 0x638d268 <line:7:5, col:16>
|     `-VarDecl 0x638d210 <col:5, col:15> C 'int [1000]'
|-FunctionDecl 0x638d7b0 <line:13:1, line:19:1> foo2 'void ()'
| `-CompoundStmt 0x638dc18 <line:13:12, line:19:1>
|   |-DeclStmt 0x638d8e8 <line:14:5, col:16>
|   | `-VarDecl 0x638d890 <col:5, col:15> A 'int [1000]'
|   |-DeclStmt 0x638d9d8 <line:15:5, col:17>
|   | `-VarDecl 0x638d980 <col:5, col:16> B 'char [1000]'
|   `-DeclStmt 0x638da88 <line:16:5, col:16>
|     `-VarDecl 0x638da30 <col:5, col:15> C 'int [1000]'
|-FunctionDecl 0x638dc70 <line:22:1, line:29:1> foo3 'void ()'
| `-CompoundStmt 0x638e0c0 <line:22:12, line:29:1>
|   |-DeclStmt 0x638dda8 <line:23:5, col:17>
|   | `-VarDecl 0x638dd50 <col:5, col:16> A 'char [1000]'
|   |-DeclStmt 0x638de58 <line:24:5, col:17>
|   | `-VarDecl 0x638de00 <col:5, col:16> B 'char [1000]'
```

```
|     -DeclStmt 0x638df08 <line:25:5, col:17>
|       `-VarDecl 0x638deb0 <col:5, col:16> C 'char [1000]'
`-FunctionDecl 0x638e110 <line:32:1, line:41:1> foo4 'void ()'
  `-CompoundStmt 0x638e720 <line:32:12, line:41:1>
    |-DeclStmt 0x638e2d8 <line:33:5, col:19>
    | `-VarDecl 0x638e280 <col:5, col:18> A 'double [1000]'
    |-DeclStmt 0x638e388 <line:34:5, col:19>
    | `-VarDecl 0x638e330 <col:5, col:18> B 'double [1000]'
    |-DeclStmt 0x638e438 <line:35:5, col:19>
    | `-VarDecl 0x638e3e0 <col:5, col:18> C 'double [1000]'
    `-DeclStmt 0x638e6b8 <line:39:5, col:19>
      `-VarDecl 0x638e660 <col:5, col:18> D 'char [1000]'
11 errors generated.
[testing] ./PR002/test/P2_testcase_12.c
```

实验证明，对于（无pragma）类型不匹配，非int型数组，数组长度不匹配的情况不能生成AST，同时会报错。