# 编译原理研讨课实验PR002实验报告

## 任务说明

扩展C99语言标准，使得添加了elementWise制导定义的函数能够支持整形静态数组的向量化操作，并构建新的AST

- 只有被`#pragma elementWise`标注的函数才支持向量化操作
- 支持加法、乘法、赋值三种操作
- 支持C语言标准的int类型
- 操作数应为静态大小的一维数组
- 操作前必须进行类型检查（类型相同的静态数组）以及大小匹配（数组大小相等）
- 不能破坏原有C语言代码的语义

## 成员组成


- 蔡润泽 2017K8009908018
- 燕澄皓 2017K8009915041

# 实验设计

## 设计思路

本次实验要对int类型的数组进行运算操作，编译器要调用`Sema::ActOnBinOp`对二元运算符进行分析。

在`Sema::ActOnBinOp`中调用`ExprResult Sema::BuildBinOp`创建非重载二元操作，通过在`Sema::BuildBinOp`调用`ExprResult Sema::CreateBuiltinBinOp`实现。在这个函数之中根据opc处理实验所要求的三种运算：

- **赋值: "="**
  处理`=`的时候主函数`CheckAssignmentOperands`会调用检查类型的函数`CheckSingleAssignmentConstraints`。为了支持数组赋值操作，我们需要对该函数进行修改，使得对于合乎我们要求的数组赋值操作，检查也返回`compatible`的结果。
  注意：在进行赋值操作时需要检查左侧是否是`modifiable`的左值，因为数组不是传统的可修改的左值，在没有添加`elementWise`制导的时候调用到`CheckForModifiableLvalue`时会进行报错处理。因此在我们添加制导之后，需要让其直接返回`false`而不做更多其他的处理。
- **加法: "+"**
  处理`+`的时候会编译器会调用`QualType Sema::CheckAdditionOperands`。为了支持数组的加法操作，当处理到`elementWise`制导下的数组加法操作时，我们需要在这个函数中添加对加法运算符两个操作数的类型匹配，大小检查等处理。
- **乘法: "*"**
  处理`*`的时候编译器会调用`QualType Sema::CheckMultiplyDivideOperands`。为了支持数组的乘法操作，当处理到`elementWise`制导下的数组乘法操作时，我们需要在这个函数中添加对乘法运算符两个操作数的类型匹配、大小检查等处理。

由于我们只支持对整形静态数组进行向量化的操作，需要添加判断类型是否为`Int`的函数。

综上，本次实验需要修改或添加的函数如下：

```
//modify
Sema::CheckSingleAssignmentConstraints
static bool CheckForModifiableLvalue
QualType Sema::CheckAdditionOperands
QualType Sema::CheckMultiplyDivideOperands

//add
inline bool Type::isIntType() const;
```

## 实验实现

- **添加判断类型是否为`Int`的类型检查函数**

  修改文件位于`llvm/tools/clang/include/clang/AST/Type.h`，代码实现如下：

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

- **修改`Sema::CheckSingleAssignmentConstraints`函数**

  数组并非传统的可修改的左值（即`modifiableLvalue`），当包含`elementWise`制导的时候，需要修改代码以跳过原先的报错处理流程。
  修改文件位于`llvm/tools/clang/lib/Sema/SemaExpr.cpp`，代码改动如下：

```
case Expr::MLV_ArrayTemporary:
    if(S.ElementWiseOn) return false;           //lab2 added
    Diag = diag::err_typecheck_array_not_modifiable_lvalue;
    NeedType = true;
    break;
```

  添加对赋值号"="两侧表达式进行类型检查的操作，大体思路如下：
  - 如果包含`elementWise`制导且表达式左右均为常数组类型，获取左右两边表达式的未限定规范类型以及数组元素的类型。
  - 如果赋值号两侧表达式的未限定规范类型相匹配，并且数组元素为`Int`，返回`Compatible`的结果。
  - 注意，数组元素为`Int`类型的特判说明了我们只支持对一维静态整形数组的向量化操作。

  修改文件位于`llvm/tools/clang/lib/Sema/SemaExpr.cpp`，代码实现如下：

```
if (this->ElementWiseOn) {
  QualType RHSType = RHS.get()->getType();

  if (ConstantArrayType::classof(LHSType.getTypePtr()) &&
      ConstantArrayType::classof(RHSType.getTypePtr())){
    // extend elementWise operations
    ConstantArrayType * RHSArrayType = dyn_cast<ConstantArrayType>(RHStype);
    QualType RHSElementType = RHSArrayType->getElementType().getUnqualifiedType();

    LHSType = Context.getCanonicalType(LHSType).getUnqualifiedType();
    RHSType = Context.getCanonicalType(RHStype).getUnqualifiedType();

    if (LHSType == RHSType && RHSElementType -> isIntType()) {
      return Compatible;
    } else {
      return Incompatible;
    }
  }
}
```

- **修改`QualType Sema::CheckAdditionOperands`函数：**

  当包含`elementWise`制导定义的时候，我们首先需要对操作的数组进行类型检查。这里基本的处理流程大致同赋值操作，如果左右两边表达式的未限定规范类型相同，且数组元素均为`Int`，并

且数组大小也相同，则通过了加法操作的类型检查。
类型检查通过后我们要进一步核查左右表达式的左值右值的性质。对于复杂表达式，在运算过程中如果得到的中间的表达式结果不为右值，那么我们就需要将其转化为可进行加法操作的右值，并构建新的运算符等信息的`AST`节点。
修改文件位于`llvm/tools/clang/lib/Sema/SemaExpr.cpp`，代码实现如下：

```
if (this->ElementWiseOn) {
  const Type *LHSType = LHS.get()->getType().getTypePtr();
  const Type *RHSType = RHS.get()->getType().getTypePtr();
  if (ConstantArrayType::classof(LHSType) && ConstantArrayType::classof(RHSType)) {
    const ConstantArrayType *LHSArrayType = dyn_cast<ConstantArrayType>(LHSType);
    const ConstantArrayType *RHSArrayType = dyn_cast<ConstantArrayType>(RHSType);
    llvm::APInt LHSArrayLen = LHSArrayType->getSize();
    llvm::APInt RHSArrayLen = RHSArrayType->getSize();
    QualType LHSElementType = LHSArrayType->getElementType().getUnqualifiedType();
    QualType RHSElementType = RHSArrayType->getElementType().getUnqualifiedType();
    if (LHSArrayLen == RHSArrayLen && LHSElementType == RHSElementType && LHSElemen
      Qualifiers qual;
      if (!LHS.get()->isRValue()) {
        // Cast to RValue
        LHS = ImplicitCastExpr::Create(Context, Context.getUnqualifiedArrayType(LHS
      }
      if (!RHS.get()->isRValue()) {
        // Cast to RValue
        RHS = ImplicitCastExpr::Create(Context, Context.getUnqualifiedArrayType(RHS
      }
      return LHS.get()->getType();
    }
  }
}
```

- **修改`QualType Sema::CheckMultiplyDivideOperands`函数**

  处理思路同加法。
  修改文件位于`llvm/tools/clang/lib/Sema/SemaExpr.cpp`，代码实现如下：

```
if (this->ElementWiseOn) {
  Expr *LHS.get() = LHS.get(), *RHS.get() = RHS.get();
  const Type *LHSType = LHS.get()->getType().getTypePtr();
  const Type *RHSType = RHS.get()->getType().getTypePtr();
  if (ConstantArrayType::classof(LHSType) && ConstantArrayType::classof(RHSType)) {
    const ConstantArrayType *LHSArrayType = dyn_cast<ConstantArrayType>(LHSType);
    const ConstantArrayType *RHSArrayType = dyn_cast<ConstantArrayType>(RHSType);
    llvm::APInt LHSArrayLen = LHSArrayType->getSize();
    llvm::APInt RHSArrayLen = RHSArrayType->getSize();
    QualType LHSElementType = LHSArrayType->getElementType().getUnqualifiedType();
    QualType RHSElementType = RHSArrayType->getElementType().getUnqualifiedType();
    if (LHSArrayLen == RHSArrayLen && LHSElementType == RHSElementType && LHSElemen
      Qualifiers qual;
      if (!LHS.get()->isRValue()) {
        // Cast to RValue
        LHS = ImplicitCastExpr::Create(Context, Context.getUnqualifiedArrayType(LHS
      }
      if (!RHS.get()->isRValue()) {
        // Cast to RValue
        RHS = ImplicitCastExpr::Create(Context, Context.getUnqualifiedArrayType(RHS
      }
      return LHS.get()->getType();
    }
  }
}
```

# 总结

## 实验结果总结

对于任务书上需要验证的几个Case，本实验的结果如下：

### CASE 1

测试代码如下：

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

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
TranslationUnitDecl 0x6835de0 <<invalid sloc>>
|-TypedefDecl 0x68362c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6836320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6836670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6836710 <test.c:2:1, line:9:1> foo1 'void ()'
  `-CompoundStmt 0x68633b0 <line:2:12, line:9:1>
    |-DeclStmt 0x6836888 <line:3:5, col:16>
    | `-VarDecl 0x6836830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x6836938 <line:4:5, col:16>
    | `-VarDecl 0x68368e0 <col:5, col:15> B 'int [1000]'
    |-DeclStmt 0x68369e8 <line:5:5, col:16>
    | `-VarDecl 0x6836990 <col:5, col:15> C 'int [1000]'
    |-BinaryOperator 0x6836ad0 <line:6:5, col:13> 'int [1000]' '='
    | |-DeclRefExpr 0x6836a00 <col:5> 'int [1000]' lvalue Var 0x6836990 'C' 'int [1000]'
    | `-BinaryOperator 0x6836aa8 <col:9, col:13> 'int [1000]' '+'
    |   |-ImplicitCastExpr 0x6836a78 <col:9> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x6836a28 <col:9> 'int [1000]' lvalue Var 0x6836830 'A' 'int [1000]'
    |   `-ImplicitCastExpr 0x6836a90 <col:13> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x6836a50 <col:13> 'int [1000]' lvalue Var 0x68368e0 'B' 'int [1000]'
    |-BinaryOperator 0x6863310 <line:7:5, col:13> 'int [1000]' '='
    | |-DeclRefExpr 0x6863240 <col:5> 'int [1000]' lvalue Var 0x6836990 'C' 'int [1000]'
    | `-BinaryOperator 0x68632e8 <col:9, col:13> 'int [1000]' '*'
    |   |-ImplicitCastExpr 0x68632b8 <col:9> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x6863268 <col:9> 'int [1000]' lvalue Var 0x6836830 'A' 'int [1000]'
    |   `-ImplicitCastExpr 0x68632d0 <col:13> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x6863290 <col:13> 'int [1000]' lvalue Var 0x68368e0 'B' 'int [1000]'
    `-BinaryOperator 0x6863388 <line:8:5, col:9> 'int [1000]' '='
      |-DeclRefExpr 0x6863338 <col:5> 'int [1000]' lvalue Var 0x6836990 'C' 'int [1000]'
      `-DeclRefExpr 0x6863360 <col:9> 'int [1000]' lvalue Var 0x6836830 'A' 'int [1000]'
0
```

该测试样例符合element wise的标准，因此最后的输出结果为`0`。

## CASE 2

测试代码如下：

```
#pragma elementWise
void foo10(){
    int A[1000];
    int B[1000];
    const 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);
}
```

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
TranslationUnitDecl 0x61afde0 <<invalid sloc>>
|-TypedefDecl 0x61b02c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x61b0320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x61b0670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x61b0710 <test.c:12:1, line:22:1> foo10 'void ()'
  `-CompoundStmt 0x61dd9e0 <line:12:13, line:22:1>
    |-DeclStmt 0x61b0888 <line:13:5, col:16>
    | `-VarDecl 0x61b0830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x61b0938 <line:14:5, col:16>
    | `-VarDecl 0x61b08e0 <col:5, col:15> B 'int [1000]'
    |-DeclStmt 0x61b0a28 <line:15:5, col:22>
    | `-VarDecl 0x61b09d0 <col:5, col:21> C 'const int [1000]'
    |-DeclStmt 0x61b0ad8 <line:16:5, col:16>
    | `-VarDecl 0x61b0a80 <col:5, col:15> D 'int [1000]'
    |-BinaryOperator 0x61dd368 <line:17:5, col:17> 'int [1000]' '='
    | |-DeclRefExpr 0x61dd230 <col:5> 'int [1000]' lvalue Var 0x61b0a80 'D' 'int [1000]'
    | `-BinaryOperator 0x61dd340 <col:9, col:17> 'int [1000]' '+'
    |   |-BinaryOperator 0x61dd2d8 <col:9, col:13> 'int [1000]' '+'
    |   | |-ImplicitCastExpr 0x61dd2a8 <col:9> 'int [1000]' <LValueToRValue>
    |   | | `-DeclRefExpr 0x61dd258 <col:9> 'int [1000]' lvalue Var 0x61b0830 'A' 'int [1000]'
    |   | `-ImplicitCastExpr 0x61dd2c0 <col:13> 'int [1000]' <LValueToRValue>
    |   |   `-DeclRefExpr 0x61dd280 <col:13> 'int [1000]' lvalue Var 0x61b08e0 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x61dd328 <col:17> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x61dd300 <col:17> 'const int [1000]' lvalue Var 0x61b09d0 'C' 'const int [1000]'
    |-BinaryOperator 0x61dd4c8 <line:18:5, col:17> 'int [1000]' '='
    | |-DeclRefExpr 0x61dd390 <col:5> 'int [1000]' lvalue Var 0x61b0a80 'D' 'int [1000]'
    | `-BinaryOperator 0x61dd4a0 <col:9, col:17> 'int [1000]' '+'
    |   |-BinaryOperator 0x61dd438 <col:9, col:13> 'int [1000]' '*'
    |   | |-ImplicitCastExpr 0x61dd408 <col:9> 'int [1000]' <LValueToRValue>
    |   | | `-DeclRefExpr 0x61dd3b8 <col:9> 'int [1000]' lvalue Var 0x61b0830 'A' 'int [1000]'
    |   | `-ImplicitCastExpr 0x61dd420 <col:13> 'int [1000]' <LValueToRValue>
    |   |   `-DeclRefExpr 0x61dd3e0 <col:13> 'int [1000]' lvalue Var 0x61b08e0 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x61dd488 <col:17> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x61dd460 <col:17> 'const int [1000]' lvalue Var 0x61b09d0 'C' 'const int [1000]'
    |-BinaryOperator 0x61dd630 <line:19:5, col:19> 'int [1000]' '='
    | |-DeclRefExpr 0x61dd4f0 <col:5> 'int [1000]' lvalue Var 0x61b0a80 'D' 'int [1000]'
    | `-ParenExpr 0x61dd610 <col:9, col:19> 'int [1000]'
    |   `-BinaryOperator 0x61dd5e8 <col:10, col:18> 'int [1000]' '='
    |     |-DeclRefExpr 0x61dd518 <col:10> 'int [1000]' lvalue Var 0x61b0a80 'D' 'int [1000]'
    |     `-BinaryOperator 0x61dd5c0 <col:14, col:18> 'int [1000]' '+'
    |       |-ImplicitCastExpr 0x61dd590 <col:14> 'int [1000]' <LValueToRValue>
    |       | `-DeclRefExpr 0x61dd540 <col:14> 'int [1000]' lvalue Var 0x61b0830 'A' 'int [1000]'
    |       `-ImplicitCastExpr 0x61dd5a8 <col:18> 'int [1000]' <LValueToRValue>
    |         `-DeclRefExpr 0x61dd568 <col:18> 'int [1000]' lvalue Var 0x61b08e0 'B' 'int [1000]'
    |-BinaryOperator 0x61dd7b0 <line:20:5, col:19> 'int [1000]' '='
    | |-DeclRefExpr 0x61dd658 <col:5> 'int [1000]' lvalue Var 0x61b0a80 'D' 'int [1000]'
    | `-BinaryOperator 0x61dd788 <col:9, col:19> 'int [1000]' '*'
    |   |-ParenExpr 0x61dd728 <col:9, col:15> 'int [1000]'
    |   | `-BinaryOperator 0x61dd700 <col:10, col:14> 'int [1000]' '+'
    |   |   |-ImplicitCastExpr 0x61dd6d0 <col:10> 'int [1000]' <LValueToRValue>
    |   |   | `-DeclRefExpr 0x61dd680 <col:10> 'int [1000]' lvalue Var 0x61b0830 'A' 'int [1000]'
    |   |   `-ImplicitCastExpr 0x61dd6e8 <col:14> 'int [1000]' <LValueToRValue>
    |   |     `-DeclRefExpr 0x61dd6a8 <col:14> 'int [1000]' lvalue Var 0x61b08e0 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x61dd770 <col:19> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x61dd748 <col:19> 'const int [1000]' lvalue Var 0x61b09d0 'C' 'const int [1000]'
    `-BinaryOperator 0x61dd9b8 <line:21:5, col:25> 'int [1000]' '='
      |-DeclRefExpr 0x61dd7d8 <col:5> 'int [1000]' lvalue Var 0x61b0a80 'D' 'int [1000]'
```

 case2-2

该测试样例符合element wise的标准，因此最后的输出结果为`0`。

## CASE 3

测试代码如下：

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

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
test.c:28:11: error: invalid operands to binary
      expression ('int *' and 'int *')
    C = A + B;
        ~ ^ ~
test.c:29:11: error: invalid operands to binary
      expression ('int *' and 'int *')
    C = A * B;
        ~ ^ ~
test.c:30:7: error: array type 'int [1000]' is not
      assignable
    C = A;
    ~ ^
TranslationUnitDecl 0x5c20de0 <<invalid sloc>>
|-TypedefDecl 0x5c212c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5c21320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5c21670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5c21710 <test.c:24:1, line:31:1> foo2 'void ()'
  `-CompoundStmt 0x5c4e658 <line:24:12, line:31:1>
    |-DeclStmt 0x5c21888 <line:25:5, col:16>
    | `-VarDecl 0x5c21830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x5c21938 <line:26:5, col:16>
    | `-VarDecl 0x5c218e0 <col:5, col:15> B 'int [1000]'
    `-DeclStmt 0x5c219e8 <line:27:5, col:16>
      `-VarDecl 0x5c21990 <col:5, col:15> C 'int [1000]'
3 errors generated.
1
```

因为没有加入`#pragma elementWise`，该测试样例不符合element wise的标准。

因此最后的输出结果为`1`。

## CASE 4

测试代码如下：

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

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
test.c:39:7: error: assigning to 'int [1000]' from incompatible type 'int *'
    C = D;
      ^ ~
TranslationUnitDecl 0x606ede0 <<invalid sloc>>
|-TypedefDecl 0x606f2c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x606f320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x606f670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x606f710 <test.c:34:1, line:40:1> foo3 'void ()'
  `-CompoundStmt 0x609c5b0 <line:34:12, line:40:1>
    |-DeclStmt 0x606f888 <line:35:5, col:16>
    | `-VarDecl 0x606f830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x606f938 <line:36:5, col:16>
    | `-VarDecl 0x606f8e0 <col:5, col:15> B 'int [1000]'
    |-DeclStmt 0x606f9e8 <line:37:5, col:16>
    | `-VarDecl 0x606f990 <col:5, col:15> C 'int [1000]'
    `-DeclStmt 0x606fa98 <line:38:5, col:11>
      `-VarDecl 0x606fa40 <col:5, col:10> D 'int *'
1 error generated.
1
```

因为D的变量类型不能直接赋值给数组变量C，该测试样例为不合法的赋值语句。

因此最后的输出结果为`1`。

## CASE 5

测试代码如下：

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

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
test.c:48:13: error: expression is not assignable
    (A + B) = C;
    ~~~~~~~ ^
TranslationUnitDecl 0x5d8ade0 <<invalid sloc>>
|-TypedefDecl 0x5d8b2c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5d8b320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5d8b670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5d8b710 <test.c:43:1, line:49:1> foo4 'void ()'
  `-CompoundStmt 0x5db82e0 <line:43:12, line:49:1>
    |-DeclStmt 0x5d8b888 <line:44:5, col:16>
    | `-VarDecl 0x5d8b830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x5d8b938 <line:45:5, col:16>
    | `-VarDecl 0x5d8b8e0 <col:5, col:15> B 'int [1000]'
    |-DeclStmt 0x5d8b9e8 <line:46:5, col:16>
    | `-VarDecl 0x5d8b990 <col:5, col:15> C 'int [1000]'
    `-DeclStmt 0x5d8ba98 <line:47:5, col:11>
      `-VarDecl 0x5d8ba40 <col:5, col:10> D 'int *'
1 error generated.
1
```

因为C不能直接赋值给数组变量A+B，该测试样例为不合法的赋值语句。

因此最后的输出结果为`1`。

## CASE 6

测试代码如下：

```
#pragma elementWise
void foo5(){
    int A[1000];
    int B[1000];
    int C[1000];
    int E[10001];
    E = A;
    E = A + B;
    E = A * B;
}
```

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
test.c:57:7: error: assigning to 'int [10001]' from incompatible type 'int [1000]'
    E = A;
      ^ ~
test.c:58:7: error: assigning to 'int [10001]' from incompatible type 'int [1000]'
    E = A + B;
      ^ ~~~~~
test.c:59:7: error: assigning to 'int [10001]' from incompatible type 'int [1000]'
    E = A * B;
      ^ ~~~~~
TranslationUnitDecl 0x5555de0 <<invalid sloc>>
|-TypedefDecl 0x55562c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5556320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5556670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5556710 <test.c:52:1, line:60:1> foo5 'void ()'
  `-CompoundStmt 0x5583780 <line:52:12, line:60:1>
    |-DeclStmt 0x5556888 <line:53:5, col:16>
    | `-VarDecl 0x5556830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x5556938 <line:54:5, col:16>
    | `-VarDecl 0x55568e0 <col:5, col:15> B 'int [1000]'
    |-DeclStmt 0x55569e8 <line:55:5, col:16>
    | `-VarDecl 0x5556990 <col:5, col:15> C 'int [1000]'
    `-DeclStmt 0x5556ad8 <line:56:5, col:17>
      `-VarDecl 0x5556a80 <col:5, col:16> E 'int [10001]'
3 errors generated.
1
```

因为数组变量E的长度和A、B、C的长度不一致，因此A、B、C的相关运算结果不能赋值给数组变量E，该测试样例为不合法的赋值语句。

因此最后的输出结果为`1`。

## CASE 7

测试代码如下：

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

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
test.c:67:11: error: invalid operands to binary expression ('char *' and 'int *')
    C = A + B;
        ~ ^ ~
test.c:68:11: error: invalid operands to binary expression ('char *' and 'int *')
    C = A * B;
        ~ ^ ~
test.c:69:7: error: assigning to 'int [1000]' from incompatible type 'char [1000]'
    C = A;
      ^ ~
TranslationUnitDecl 0x6598de0 <<invalid sloc>>
|-TypedefDecl 0x65992c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6599320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6599670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6599710 <test.c:63:1, line:70:1> foo6 'void ()'
  `-CompoundStmt 0x65c66c8 <line:63:12, line:70:1>
    |-DeclStmt 0x6599888 <line:64:5, col:17>
    | `-VarDecl 0x6599830 <col:5, col:16> A 'char [1000]'
    |-DeclStmt 0x6599978 <line:65:5, col:16>
    | `-VarDecl 0x6599920 <col:5, col:15> B 'int [1000]'
    `-DeclStmt 0x6599a28 <line:66:5, col:16>
      `-VarDecl 0x65999d0 <col:5, col:15> C 'int [1000]'
3 errors generated.
1
```

因为数组变量A的基本变量类型为char和B、C的int基本变量类型不一致，因此A与B、C的不能进行相关运算和赋值，该测试样例为不合法的语句。

因此最后的输出结果为`1`。

## CASE 8

测试代码如下：

```
#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;
}
```

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
test.c:79:7: error: assigning to 'int [10][100]' from incompatible type 'int [1000]'
    E = A;
      ^ ~
test.c:80:7: error: assigning to 'int [10][100]' from incompatible type 'int [1000]'
    E = A + B;
      ^ ~~~~~
test.c:81:7: error: assigning to 'int [10][100]' from incompatible type 'int [1000]'
    E = A * B;
      ^ ~~~~~
TranslationUnitDecl 0x55c0de0 <<invalid sloc>>
|-TypedefDecl 0x55c12c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x55c1320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x55c1670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x55c1710 <test.c:73:1, line:82:1> foo7 'void ()'
  `-CompoundStmt 0x55ee890 <line:73:12, line:82:1>
    |-DeclStmt 0x55c1888 <line:74:5, col:16>
    | `-VarDecl 0x55c1830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x55c1938 <line:75:5, col:16>
    | `-VarDecl 0x55c18e0 <col:5, col:15> B 'int [1000]'
    |-DeclStmt 0x55c19e8 <line:76:5, col:16>
    | `-VarDecl 0x55c1990 <col:5, col:15> C 'int [1000]'
    |-DeclStmt 0x55c1a98 <line:77:5, col:11>
    | `-VarDecl 0x55c1a40 <col:5, col:10> D 'int *'
    `-DeclStmt 0x55ee338 <line:78:5, col:19>
      `-VarDecl 0x55ee2e0 <col:5, col:18> E 'int [10][100]'
3 errors generated.
1
```

因为数组变量E是一个二维数组，而A、B、C是一位数组，因此A、B、C的进行相关运算结果不能赋值给E，该测试样例为不合法的赋值语句。

因此最后的输出结果为`1`。

## CASE 9

测试代码如下：

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

测试结果如下：

```
llvm1@teacher-PowerEdge-M640:~$ ~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c; echo $?
test.c:89:7: error: read-only variable is not assignable
    C = A;
    ~ ^
test.c:90:7: error: read-only variable is not assignable
    C = A + B;
    ~ ^
TranslationUnitDecl 0x5a51de0 <<invalid sloc>>
|-TypedefDecl 0x5a522c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5a52320 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5a52670 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5a52710 <test.c:85:1, line:91:1> foo8 'void ()'
  `-CompoundStmt 0x5a7f298 <line:85:12, line:91:1>
    |-DeclStmt 0x5a52888 <line:86:5, col:16>
    | `-VarDecl 0x5a52830 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x5a52938 <line:87:5, col:16>
    | `-VarDecl 0x5a528e0 <col:5, col:15> B 'int [1000]'
    `-DeclStmt 0x5a52a28 <line:88:5, col:22>
      `-VarDecl 0x5a529d0 <col:5, col:21> C 'const int [1000]'
2 errors generated.
1
```

因为数组变量C是一个常量数组，因此A、B以及其相关运算结果不能赋值给C，该测试样例为不合法的赋值语句。

因此最后的输出结果为`1`。

# 分成员总结

## 蔡润泽

本次实验主要负责编写代码以及检查测试结果。本次实验的代码量比较小，涉及到的clang文件也比较少。需要完成的工作非常的直观，即查错以及生成抽象语法树。

然而本次实验也有一些挑战，因为任务书的内容不像第一次实验那样手把手指导步骤，因此需要我们阅读clang工程文件里的许多有着类似处理步骤的代码，模仿并学习一些其他函数的使用。需要花费一些时间来理清不同对象以及函数的使用层次和使用规则。

## 燕澄皓

本次实验中主要负责调试代码以及撰写实验报告。

此次实验上手较难，但是理清具体要做的事情后实现又较为简单，总体而言难度适中，实验的代码量小但是精。个人认为实现起来比较让人困惑的点在于对运算对象的结构类型处理。类型的层层转化以及后续对于左值右值的处理，需要我们在实现时始终保持头脑清醒，明确每一步做到了哪里，接下来还要怎么做。