# Static Addresses

A static expression is an expression that is located at a specific place in the memory. The address can be identified by its name and offset. In the following example where i and p are static variables, p is static, and the linker eventually decides its value.

int i;

int \*p = &i;

### Static Value and Address

The StaticValue and StaticAddress classes are identical sub classes of StaticBase, which holds a name and an offset. In the end, only static addresses are allowed, but static values can hold intermediate values during the parsing. For instance, in the static address &a[3], a[3] is temporary stored as a static value.

StaticBase.cs

namespace CCompiler {

public abstract class StaticBase {

private string m\_uniqueName;

private int m\_offset;

public StaticBase(string name, int offset) {

m\_uniqueName = name;

m\_offset = offset;

}

public string UniqueName {

get { return m\_uniqueName; }

}

public int Offset {

get { return m\_offset; }

}

public override string ToString() {

if (m\_offset > 0) {

return m\_uniqueName + " + " + m\_offset;

}

else if (m\_offset < 0) {

return m\_uniqueName + " - " + (-m\_offset);

}

else {

return m\_uniqueName;

}

}

}

public class StaticValue : StaticBase {

public StaticValue(string name, int offset)

:base(name, offset) {

// Empty.

}

}

public class StaticAddress : StaticBase {

public StaticAddress(string name, int offset)

:base(name, offset) {

// Empty.

}

}

}

### Static Expression

The StaticExpression class holds the two methods Binary and Unary, which takes a binary or unary expression and returns a static expression if there is one, or null otherwise.

StaticExpression.cs

using System.Numerics;

namespace CCompiler {

public class StaticExpression {

The Binary method exams the expressions if the operator is binary addition or subtraction, index, or dot.

public static Expression Binary(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

object leftValue = leftExpression.Symbol.Value,

rightValue = rightExpression.Symbol.Value;

In the addition case, the operand values must be a static address and a constant integer value, or an extern or static array and a constant value. For instance, &i + 2, 2 + &i, a + 2 or 2 + a, where i an integer and a is an array. In case of static address and an integral value on either side, we call GenerateAddition to generate the resulting static address.

if ((leftValue is StaticAddress) && // &i + 2

(rightValue is BigInteger)) {

return GenerateAddition(leftSymbol, (BigInteger) rightValue);

}

else if ((leftValue is BigInteger) && // 2 + &i

(rightValue is StaticAddress)){

return GenerateAddition(rightSymbol, (BigInteger) leftValue);

}

In case of a static or extern array and an integer value on either side, we also call GenerateAddition.

else if (leftSymbol.IsExternOrStaticArray() && // a + 2

(rightValue is BigInteger)) {

return GenerateAddition(leftSymbol, (BigInteger)rightValue);

}

else if ((leftValue is BigInteger) && // 2 + a

rightSymbol.IsExternOrStaticArray()) {

return GenerateAddition(rightSymbol, (BigInteger) leftValue);

}

break;

In the subtraction case the left operand must a static address or a static or extern array, and the right operand must be a constant integer value. For instance, &i - 2 or a - 2. Unlik the addition case above, we cannot swap the operands, the 2 - &i and 2 - a cases are not allowed.

case MiddleOperator.Subtract:

if ((leftValue is StaticAddress) && // &i - 2

(rightValue is BigInteger)) {

return GenerateAddition(leftSymbol,

-((BigInteger) rightValue));

}

else if (leftSymbol.IsExternOrStaticArray() && // a - 2

(rightValue is BigInteger)) {

return GenerateAddition(leftSymbol,

-((BigInteger) rightValue));

}

break;

In the index case, the operands must be a static address or a static or extern array, and a constant integer value, on either side. For instance, &i[2] and a[2] as well as 2[&i] are 2[a] are allowed. We call GenerateIndex to generate the static address.

case MiddleOperator.Index:

if ((leftValue is StaticAddress) && (rightValue is BigInteger)){

return GenerateIndex(leftSymbol, (BigInteger) rightValue);

}

else if ((leftValue is BigInteger) &&

(rightValue is StaticAddress)){

return GenerateIndex(rightSymbol, (BigInteger) leftValue);

}

else if (leftSymbol.IsExternOrStaticArray() &&

(rightValue is BigInteger)) {

return GenerateIndex(leftSymbol, (BigInteger) rightValue);

}

else if ((leftValue is BigInteger) &&

rightSymbol.IsExternOrStaticArray()) {

return GenerateIndex(rightSymbol, (BigInteger) leftValue);

}

break;

In the dot case, the operands must be an extern or static struct or union, or a static address. For instance, s.i where s is a static or extern struct and i is one of its members. Note that the resulting value is an object of the StaticValue class rather than the StaticAddress class.

case MiddleOperator.Dot:

if (leftSymbol.IsExternOrStatic()) {

object resultValue =

new StaticValue(leftSymbol.UniqueName, rightSymbol.Offset);

Symbol resultSymbol = new Symbol(leftType, resultValue);

return (new Expression(resultSymbol, null, null));

}

break;

}

return null;

}

The GenerateAddition method generates a static address for an addition expression.

private static Expression GenerateAddition(Symbol symbol,

BigInteger value) {

int offset = ((int) value) \* symbol.Type.PointerOrArrayType.Size();

StaticAddress resultValue;

if (symbol.Value is StaticAddress) {

StaticAddress staticAddress = (StaticAddress) symbol.Value;

resultValue = new StaticAddress(staticAddress.UniqueName,

staticAddress.Offset + offset);

}

else {

resultValue = new StaticAddress(symbol.UniqueName, offset);

}

Symbol resultSymbol = new Symbol(symbol.Type, resultValue);

return (new Expression(resultSymbol, null, null));

}

The GenerateIndex method generates the static address for an index expression.

private static Expression GenerateIndex(Symbol symbol,

BigInteger value) {

int offset = ((int) value) \* symbol.Type.ArrayType.Size();

StaticValue resultValue;

if (symbol.Value is StaticAddress) {

StaticAddress staticAddress = (StaticAddress) symbol.Value;

resultValue = new StaticValue(staticAddress.UniqueName,

staticAddress.Offset + offset);

}

else {

resultValue = new StaticValue(symbol.UniqueName, offset);

}

Symbol resultSymbol = new Symbol(symbol.Type, resultValue);

return (new Expression(resultSymbol, null, null));

}

Finally, we have to unary case. There is only one relevant operator: the address operator (‘&’).

public static Expression Unary(MiddleOperator middleOp,

Expression expression) {

Symbol symbol = expression.Symbol;

If the symbol of the address operator is a static value, we create a static address with the same name and offset. For instance, &a[i] or &s.i.

if (middleOp == MiddleOperator.Address) {

if (symbol.Value is StaticValue) { // &a[i], &s.i

StaticValue staticValue = (StaticValue) symbol.Value;

StaticAddress staticAddress =

new StaticAddress(staticValue.UniqueName, staticValue.Offset);

Symbol resultSymbol =

new Symbol(new Type(symbol.Type), staticAddress);

return (new Expression(resultSymbol, null, null));

}

If the symbol is not a static value, but holds extern or static storage, we create a static address, with the symbol name and offset zero. For instance, &i, where i holds static or extern storage.

else if (symbol.IsExternOrStatic()) {

StaticAddress staticAddress =

new StaticAddress(symbol.UniqueName, 0);

Symbol resultSymbol =

new Symbol(new Type(symbol.Type), staticAddress);

return (new Expression(resultSymbol, null, null));

}

}

return null;

}

}

}