IFCC

Generated by Doxygen 1.9.8

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

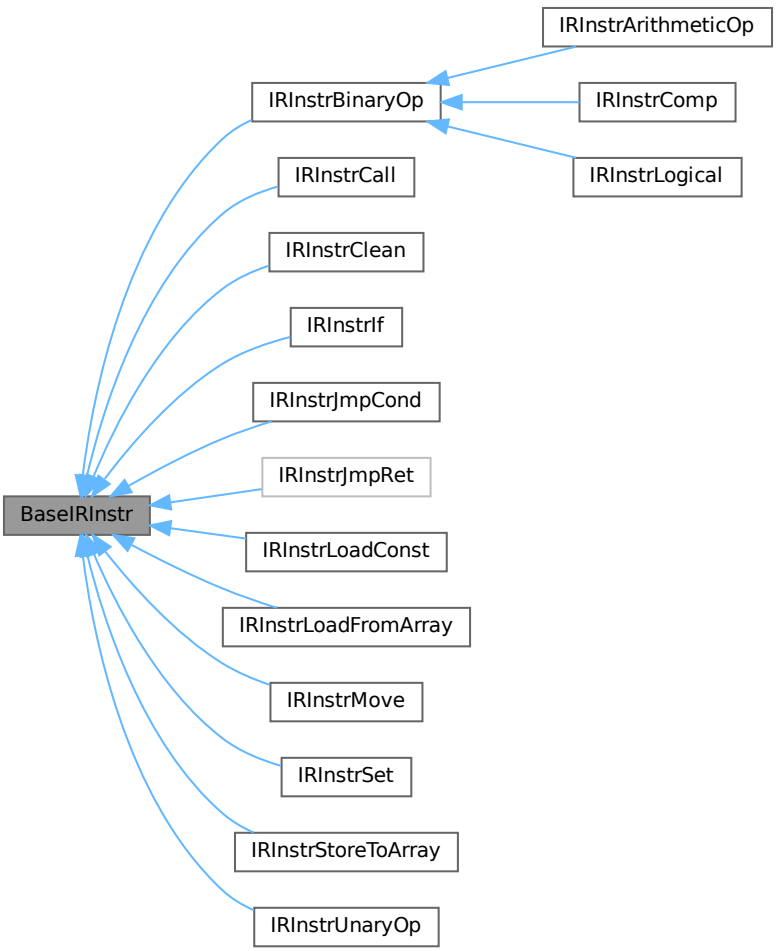
Class Documentation

3.1 BaseIRInstr Class Reference

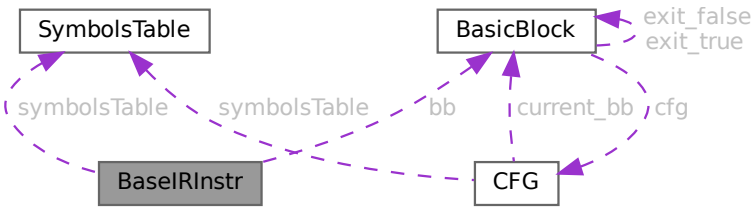
Represents a base class for intermediate representation instructions.

`#include <BaseIRInstr.h>`

Inheritance diagram for BaseIRInstr:



Collaboration diagram for BaseIRInstr:



Public Member Functions

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.
- virtual void genASM (ostream &o)=0

  Generates the assembly code for this instruction.

Protected Attributes

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.1.1 Detailed Description

Represents a base class for intermediate representation instructions.

This class serves as the base for all instruction types in the intermediate representation (IR). It provides a basic structure for handling assembly generation and access to the basic block that the instruction belongs to.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BaseIRInstr()

```
BaseIRInstr::BaseIRInstr (
            BasicBlock * bb_ )
```

Constructs an instruction for a given basic block.

Initializes the instruction with the basic block it belongs to.

Parameters

bb↩ _	The basic block to which this instruction belongs.

3.1.3 Member Function Documentation

3.1.3.1 genASM()

```
virtual void BaseIRInstr::genASM (
            ostream & o )  [pure virtual]
```

Generates the assembly code for this instruction.

This is a pure virtual function that must be implemented by derived classes to generate the specific assembly code for each type of instruction.

Parameters

o	The output stream where the generated assembly code will be written.

Implemented in IRInstrArithmeticOp, IRInstrLoadFromArray, IRInstrStoreToArray, IRInstrUnaryOp, IRInstrCall, IRInstrComp, IRInstrIf, IRInstrJmpCond, IRInstrJmpRet, IRInstrLoadConst, IRInstrLogical, IRInstrMove, and IRInstrBinaryOp.

3.1.3.2 getBB()

```
BasicBlock * BaseIRInstr::getBB ( )
```

Gets the basic block that this instruction belongs to.

Returns

The basic block associated with this instruction.

3.1.4 Member Data Documentation

3.1.4.1 bb

```
BasicBlock* BaseIRInstr::bb  [protected]
```

The basic block that this instruction belongs to.

3.1.4.2 symbolsTable

```
SymbolsTable* BaseIRInstr::symbolsTable  [protected]
```

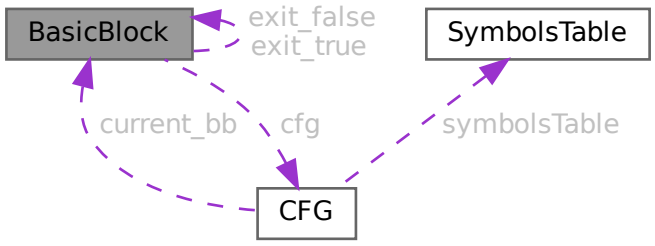
The symbol table for the current scope.

3.2 BasicBlock Class Reference

Represents a basic block in the control flow graph (CFG).

`#include <BasicBlock.h>`

Collaboration diagram for BasicBlock:



Public Member Functions

- BasicBlock (CFG *cfg, string entry_label)
  Constructs a BasicBlock with the given CFG and entry label.
- void genASM (ostream &o)
  Generates the assembly code for this basic block.
- void addIRInstr (BaseIRInstr *instr)
  Adds an instruction to the basic block.
- CFG * getCFG ()
  Gets the CFG associated with this basic block.
- string getLabel ()
  Retrieves the label associated with the current block.
- void setLabel (string label)
  Sets the label for the current block.
- vector< BaseIRInstr * > getInstr ()
  Retrieves the list of instructions within the current block.
- void setExitTrue (BasicBlock *bb)
  Sets the "true" exit point for the current block.
- void setExitFalse (BasicBlock *bb)
  Sets the "false" exit point for the current block.
- BasicBlock * getExitTrue ()
  Retrieves the "true" exit point of the current block.
- BasicBlock * getExitFalse ()
  Retrieves the "false" exit point of the current block.
- void setIsTestVar (bool isTest)
  Sets the name of the test variable.
- string getTrueLabel ()
  Get the label of the true exit point of the current block.
- string getFalseLabel ()
  Get the label of the false exit point of the current block.

Protected Attributes

- BasicBlock * exit_true
- BasicBlock * exit_false
- string label

  The label for the basic block, also used as the label in the generated assembly code.
- CFG * cfg

  The control flow graph to which this basic block belongs.
- vector< BaseIRInstr * > instrs

  A vector of instructions that belong to this basic block.
- bool is_test_var = false

  A flag indicating whether the block is a test variable.

3.2.1 Detailed Description

Represents a basic block in the control flow graph (CFG).

A basic block is a sequence of instructions with a single entry point and a single exit point. This class is responsible for managing the instructions in the block and generating the assembly code.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 BasicBlock()

```
BasicBlock::BasicBlock (
            CFG * cfg,
            string entry_label )
```

Constructs a BasicBlock with the given CFG and entry label.

Initializes a new basic block with a label and associates it with a specific control flow graph (CFG).

Parameters

cfg	The CFG where this basic block belongs.
entry_label	The entry label for the basic block.

3.2.3 Member Function Documentation

3.2.3.1 addIRInstr()

```
void BasicBlock::addIRInstr (
            BaseIRInstr * instr )
```

Adds an instruction to the basic block.

This method adds a new intermediate representation instruction (IRInstr) to the basic block.

Parameters

instr	A pointer to the instruction to add.

3.2.3.2 genASM()

```
void BasicBlock::genASM (
            ostream & o )
```

Generates the assembly code for this basic block.

This method generates the x86 assembly code for all the instructions in the basic block.

Parameters

o	The output stream where the assembly code will be written.

3.2.3.3 getCFG()

```
CFG * BasicBlock::getCFG ( )
```

Gets the CFG associated with this basic block.

This method retrieves the CFG to which this basic block belongs.

Returns

A pointer to the CFG associated with this basic block.

3.2.3.4 getExitFalse()

```
BasicBlock * BasicBlock::getExitFalse ( )
```

Retrieves the "false" exit point of the current block.

This function returns the basic block representing the "false" exit point in the control flow. It is used to identify where the program flow should continue if a condition evaluates to false.

Returns

A pointer to the `BasicBlock` representing the "false" exit.

3.2.3.5 getExitTrue()

```
BasicBlock * BasicBlock::getExitTrue ( )
```

Retrieves the "true" exit point of the current block.

This function returns the basic block representing the "true" exit point in the control flow. It is used to identify where the program flow should continue if a condition evaluates to true.

Returns

A pointer to the `BasicBlock` representing the "true" exit.

3.2.3.6 getFalseLabel()

```
string BasicBlock::getFalseLabel ( )
```

Get the label of the false exit point of the current block.

This function returns the label of the "false" exit point in the control flow. It is used to identify where the program flow should continue if a condition evaluates to false.

Returns

A string representing the label of the "false" exit.

3.2.3.7 getInstr()

```
vector< BaseIRInstr * > BasicBlock::getInstr ( )
```

Retrieves the list of instructions within the current block.

This function returns a vector containing all the instructions present in this block. Each instruction represents a basic operation in the program's flow, used in program analysis or code generation.

Returns

A vector of `BaseIRInstr *` representing the instructions in the block.

3.2.3.8 getLabel()

```
string BasicBlock::getLabel ( )
```

Retrieves the label associated with the current block.

This function returns the label associated with this particular control flow block. The label is typically used to uniquely identify this block in control flow analysis.

Returns

A string representing the label of this control flow block.

3.2.3.9 getTrueLabel()

```
string BasicBlock::getTrueLabel ( )
```

Get the label of the true exit point of the current block.

This function returns the label of the "true" exit point in the control flow. It is used to identify where the program flow should continue if a condition evaluates to true.

Returns

A string representing the label of the "true" exit.

3.2.3.10 setExitFalse()

```
void BasicBlock::setExitFalse (
            BasicBlock * bb )
```

Sets the "false" exit point for the current block.

This function sets the block that serves as the "false" exit in the control flow of the program. This is typically used for conditional branches, where the program flow follows one path if a condition is true, and another path if it is false.

Parameters

bb	A pointer to the `BasicBlock` that represents the "false" exit of the current block.

3.2.3.11 setExitTrue()

```
void BasicBlock::setExitTrue (
            BasicBlock * bb )
```

Sets the "true" exit point for the current block.

This function sets the block that serves as the "true" exit in the control flow of the program. This is often used when there is a conditional branch and the flow of execution diverges depending on the outcome of a condition.

Parameters

bb	A pointer to the `BasicBlock` that represents the "true" exit of the current block.

3.2.3.12 setIsTestVar()

```
void BasicBlock::setIsTestVar (
            bool isTest )  [inline]
```

Sets the name of the test variable.

This function sets the name of the test variable used in the basic block.

Parameters

name	The name of the test variable.

3.2.3.13 setLabel()

```
void BasicBlock::setLabel (
            string label )
```

Sets the label for the current block.

This function sets a new label for this control flow block. The label is typically used to uniquely identify this block in control flow analysis.

Parameters

label	The new label to set for this control flow block.

3.2.4 Member Data Documentation

3.2.4.1 cfg

```
CFG* BasicBlock::cfg  [protected]
```

The control flow graph to which this basic block belongs.

3.2.4.2 exit_false

```
BasicBlock* BasicBlock::exit_false  [protected]
```

Pointer to the basic block representing the "false" exit. This is used for the branch or conditional statement when the condition is false. It can be null if no "false" exit exists.

3.2.4.3 exit_true

```
BasicBlock* BasicBlock::exit_true  [protected]
```

Pointer to the basic block representing the "true" exit. This is used for the branch or conditional statement when the condition is true. It can be null if no "true" exit exists.

3.2.4.4 instrs

```
vector<BaseIRInstr *> BasicBlock::instrs  [protected]
```

A vector of instructions that belong to this basic block.

3.2.4.5 is_test_var

```
bool BasicBlock::is_test_var = false  [protected]
```

A flag indicating whether the block is a test variable.

3.2.4.6 label

```
string BasicBlock::label  [protected]
```

The label for the basic block, also used as the label in the generated assembly code.

3.3 BasicHelper Class Reference

Helper class for basic functionality related to parsing and evaluating expressions.

```
#include <BasicHelper.h>
```

Static Public Member Functions

- static optional< int > evaluateConstantExpression (ifccParser::ExprContext *ctx)

  Evaluates a constant expression represented by an AST node.

3.3.1 Detailed Description

Helper class for basic functionality related to parsing and evaluating expressions.

This class contains methods for evaluating constant expressions and performing other basic tasks related to parsing and evaluating expressions in the compiler.

3.3.2 Member Function Documentation

3.3.2.1 evaluateConstantExpression()

```
optional< int > BasicHelper::evaluateConstantExpression (
            ifccParser::ExprContext * ctx )  [static]
```

Evaluates a constant expression represented by an AST node.

Parameters

ctx	An AST node representing the constant expression.

Returns

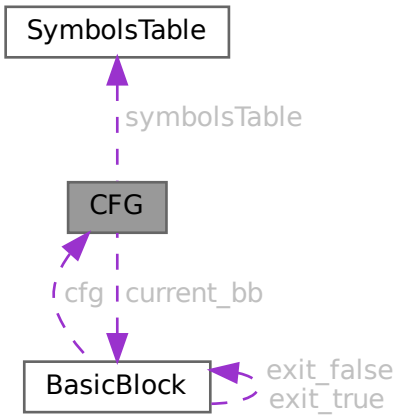
The evaluated integer value of the constant expression, or nullopt if it cannot be evaluated.

3.4 CFG Class Reference

Represents a Control Flow Graph (CFG) in an Intermediate Representation (IR).

`#include <CFG.h>`

Collaboration diagram for CFG:



Public Member Functions

- CFG (string label, SymbolsTable *symbolsTable, int initialNextFreeSymbolIndex)

  Constructs a CFG with the given label and symbol index information.
- void addBB (BasicBlock *bb)

  Adds a basic block to the control flow graph.
- void genASM (ostream &o)

  Generates the assembly code for the entire control flow graph.
- string createNewTempvar (Type t)

  Creates a new temporary variable with a unique name.
- int getVarIndex (string name)

  Retrieves the index of a variable by its name.
- Type getVarType (string name)

  Retrieves the type of a variable based on its name.
- string toRegister (string name)

  Converts a variable name into a register name.
- BasicBlock * getCurrentBasicBlock ()

  Retrieves the current basic block in the control flow graph.
- void setCurrentBasicBlock (BasicBlock *bb)

Sets the current basic block in the control flow graph.

- void resetNextFreeSymbolIndex ()

  Resets the next free symbol index to its initial value.
- string getBBName ()

  Get the next basick block name.
- void genCFGGraphviz (ostream &o)

  Generates the Graphviz representation of the control flow graph (CFG).
- string getLabel ()

  Retrieves the label associated with the control flow graph (CFG).
- void setLabel (string label)

  Sets the label for the control flow graph (CFG).
- void addWhile (BasicBlock *test, BasicBlock *body, BasicBlock *end_bb)

  Adds a while loop to the control flow graph.
- void addIfThenElse (BasicBlock *test, BasicBlock *then_bb, BasicBlock *else_bb, BasicBlock *end_bb)

  Adds an if-then-else structure to the control flow graph.
- void setSymbolsTable (SymbolsTable *symbolsTable)
- SymbolsTable * getSymbolsTable ()

Protected Attributes

- SymbolsTable * symbolsTable

  The symbols table containing information about variables and their types.
- int nextFreeSymbolIndex

  The next available symbol index.
- const int initialTempPos = 0

  The initial value for the next free symbol index.
- vector< BasicBlock * > bbs

  A vector containing all the basic blocks in the control flow graph.
- BasicBlock * current_bb

  A pointer to the current basic block being processed.
- string label

  The label associated with the control flow graph.
- int idBB

  The ID of the basic block.

3.4.1 Detailed Description

Represents a Control Flow Graph (CFG) in an Intermediate Representation (IR).

A CFG consists of basic blocks and represents the flow of control in a program. This class is responsible for managing the basic blocks and generating assembly code corresponding to the control flow graph.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 CFG()

```
CFG::CFG (
            string label,
            SymbolsTable * symbolsTable,
            int initialNextFreeSymbolIndex )
```

Constructs a CFG with the given label and symbol index information.

Initializes the control flow graph with a label and symbol index, and sets the initial index for the next free symbol.

Parameters

label	The label for the CFG.
SymbolIndex	A map that maps symbol names to their respective indices.
SymbolType	A map that maps symbol names to their respective types.
initialNextFreeSymbolIndex	The initial value for the next free symbol index.

3.4.3 Member Function Documentation

3.4.3.1 addBB()

```
void CFG::addBB (
            BasicBlock * bb )
```

Adds a basic block to the control flow graph.

This method adds a new basic block to the vector of basic blocks that make up the CFG.

Parameters

bb	A pointer to the basic block to add.

3.4.3.2 addIfThenElse()

```
void CFG::addIfThenElse (
            BasicBlock * test,
            BasicBlock * then_bb,
            BasicBlock * else_bb,
            BasicBlock * end_bb )
```

Adds an if-then-else structure to the control flow graph.

This function adds an if-then-else structure to the control flow graph by creating the necessary basic blocks for the test condition, then block, else block, and end block.

Parameters

test	The basic block representing the test condition of the if statement.
then_bb	The basic block representing the "then" branch of the if statement.
else_bb	The basic block representing the "else" branch of the if statement.
end_bb	The basic block representing the exit point of the if-then-else structure.

3.4.3.3 addWhile()

```
void CFG::addWhile (
            BasicBlock * test,
```

```
            BasicBlock * body,
            BasicBlock * end_bb )
```

Adds a while loop to the control flow graph.

This function adds a while loop to the control flow graph by creating the necessary basic blocks for the test condition, loop body, and exit block.

Parameters

test	The basic block representing the test condition of the while loop.
body	The basic block representing the body of the while loop.
end_bb	The basic block representing the exit point of the while loop.

3.4.3.4 createNewTempvar()

```
string CFG::createNewTempvar (
            Type t )
```

Creates a new temporary variable with a unique name.

This method generates a temporary variable of the specified type and returns its unique name.

Parameters

t	The type of the new temporary variable.

Returns

The name of the newly created temporary variable.

3.4.3.5 genASM()

```
void CFG::genASM (
            ostream & o )
```

Generates the assembly code for the entire control flow graph.

This method generates the assembly code for all basic blocks in the CFG.

Parameters

o	The output stream where the assembly code will be written.

3.4.3.6 genCFGGraphviz()

```
void CFG::genCFGGraphviz (
            ostream & o )
```

Generates the Graphviz representation of the control flow graph (CFG).

This function generates a Graphviz-compatible `.dot` file that visualizes the control flow of the program. The `.dot` file describes the nodes (representing basic blocks or instructions) and edges (representing the flow of control between the blocks) of the control flow graph.

The generated Graphviz representation is written to the provided output stream.

Parameters

o	The output stream to which the Graphviz `.dot` representation of the CFG is written. This is typically a file stream (e.g., `ofstream`) that writes to a `.dot` file.

3.4.3.7 getBBName()

```
string CFG::getBBName ( )
```

Get the next basick block name.

This function generates a unique name for the next basic block in the control flow graph.

Returns

A string representing the name of the next basic block.

3.4.3.8 getCurrentBasicBlock()

```
BasicBlock * CFG::getCurrentBasicBlock ( )
```

Retrieves the current basic block in the control flow graph.

This method returns a pointer to the current basic block being processed.

Returns

A pointer to the current basic block.

3.4.3.9 getLabel()

```
string CFG::getLabel ( )
```

Retrieves the label associated with the control flow graph (CFG).

This function returns a string label that represents the name or identifier associated with the current control flow graph. The label can be used to identify different parts of the program, such as functions or basic blocks.

The label is typically used for naming the nodes and edges in the Graphviz `.dot` representation or for other program analysis purposes.

Returns

A string representing the label of the control flow graph. This could be a function name, block identifier, or any other relevant label.

3.4.3.10 getSymbolsTable()

```
SymbolsTable * CFG::getSymbolsTable ( )  [inline]
```

Retrieves the symbols table associated with the control flow graph.

Returns

A pointer to the symbols table associated with the control flow graph.

3.4.3.11 getVarIndex()

```
int CFG::getVarIndex (
            string name )
```

Retrieves the index of a variable by its name.

This method retrieves the index of the variable in the symbol table.

Parameters

name	The name of the variable.

Returns

The index of the variable, or 0 if the variable does not exist.

3.4.3.12 getVarType()

```
Type CFG::getVarType (
            string name )
```

Retrieves the type of a variable based on its name.

This method looks up the variable's type from the symbols table using the provided variable name.

Parameters

name	The name of the variable whose type is to be retrieved.

Returns

The type of the specified variable.

3.4.3.13 resetNextFreeSymbolIndex()

```
void CFG::resetNextFreeSymbolIndex ( )
```

Resets the next free symbol index to its initial value.

This method resets the index for the next free symbol to its initial state.

3.4.3.14 setCurrentBasicBlock()

```
void CFG::setCurrentBasicBlock (
            BasicBlock * bb )
```

Sets the current basic block in the control flow graph.

This method sets the basic block that is currently being processed in the CFG.

Parameters

bb	A pointer to the basic block to set as the current block.

3.4.3.15 setLabel()

```
void CFG::setLabel (
            string label )
```

Sets the label for the control flow graph (CFG).

This function sets a new label for the current control flow graph. The label can be used to identify different parts of the program, such as functions or basic blocks.

Parameters

label	The new label to set for the control flow graph.

3.4.3.16 setSymbolsTable()

```
void CFG::setSymbolsTable (
            SymbolsTable * symbolsTable )
```

Sets the symbols table for the control flow graph.

Parameters

symbolsTable	A pointer to the symbols table to be set for the control flow graph.

3.4.3.17 toRegister()

```
string CFG::toRegister (
            string name )
```

Converts a variable name into a register name.

This method converts the given variable name into a corresponding register name, taking into account the position of the variable in the symbol table.

Parameters

name	The name of the variable to convert.

Returns

The register name corresponding to the input variable.

3.4.4 Member Data Documentation

3.4.4.1 bbs

```
vector<BasicBlock *> CFG::bbs  [protected]
```

A vector containing all the basic blocks in the control flow graph.

3.4.4.2 current_bb

```
BasicBlock* CFG::current_bb  [protected]
```

A pointer to the current basic block being processed.

3.4.4.3 idBB

```
int CFG::idBB  [protected]
```

The ID of the basic block.

3.4.4.4 initialTempPos

```
const int CFG::initialTempPos = 0  [protected]
```

The initial value for the next free symbol index.

3.4.4.5 label

```
string CFG::label  [protected]
```

The label associated with the control flow graph.

3.4.4.6 nextFreeSymbolIndex

```
int CFG::nextFreeSymbolIndex  [protected]
```

The next available symbol index.

3.4.4.7 symbolsTable

```
SymbolsTable* CFG::symbolsTable  [protected]
```

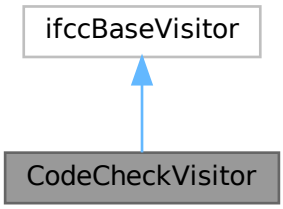
The symbols table containing information about variables and their types.

3.5 CodeCheckVisitor Class Reference

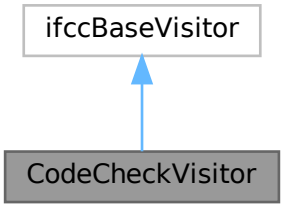
A visitor class for checking code correctness.

```
#include <CodeCheckVisitor.h>
```

Inheritance diagram for CodeCheckVisitor:



Collaboration diagram for CodeCheckVisitor:



Public Member Functions

- CodeCheckVisitor ()

  Constructs a new instance of the CodeCheckVisitor.
- virtual antlrcpp::Any visitProg (ifccParser::ProgContext ∗ctx) override

  Visits the program context in the parsed code.
- virtual antlrcpp::Any visitReturn_stmt (ifccParser::Return_stmtContext ∗ctx) override

  Visit a return statement in the parsed code.
- virtual antlrcpp::Any visitVar (ifccParser::VarContext ∗ctx) override

  Visits a variable context in the parsed code.
- virtual antlrcpp::Any visitAssign_stmt (ifccParser::Assign_stmtContext ∗ctx) override

  Visits an assignment statement in the parsed code.
- virtual antlrcpp::Any visitDecl_stmt (ifccParser::Decl_stmtContext ∗ctx) override

  Visits a declaration statement in the parsed code.
- virtual antlrcpp::Any visitExpr (ifccParser::ExprContext ∗expr)

  Visits any expression in the parsed code.
- virtual antlrcpp::Any visitAddsub (ifccParser::AddsubContext ∗ctx) override

  Visits an addition or subtraction expression.
- virtual antlrcpp::Any visitMuldiv (ifccParser::MuldivContext ∗ctx) override

  Visits a multiplication or division expression.
- virtual antlrcpp::Any visitBitwise (ifccParser::BitwiseContext ∗ctx) override

  Visits a bitwise operation expression.
- virtual antlrcpp::Any visitComp (ifccParser::CompContext ∗ctx) override

  Visits a comparison expression.
- virtual antlrcpp::Any visitUnary (ifccParser::UnaryContext ∗ctx) override

  Visits a unary expression.
- virtual antlrcpp::Any visitPost_stmt (ifccParser::Post_stmtContext ∗ctx) override

  Visits a post statement (e.g., postfix increment/decrement).
- virtual antlrcpp::Any visitPre_stmt (ifccParser::Pre_stmtContext ∗ctx) override

  Visits a pre statement (e.g., prefix increment/decrement).
- virtual antlrcpp::Any visitPre (ifccParser::PreContext ∗ctx) override

  Visits a pre-unary operation (e.g., prefix increment/decrement).
- virtual antlrcpp::Any visitPost (ifccParser::PostContext ∗ctx) override

  Visits a post-unary operation (e.g., postfix increment/decrement).
- virtual antlrcpp::Any visitDecl_func_stmt (ifccParser::Decl_func_stmtContext ∗ctx) override

  Visits a function declaration statement.
- virtual antlrcpp::Any visitCall_func_stmt (ifccParser::Call_func_stmtContext ∗ctx) override

  Visits a function call statement.
- virtual antlrcpp::Any visitBlock (ifccParser::BlockContext ∗ctx) override

  Visits a block of code in the parsed code.
- SymbolsTable ∗ getCurrentSymbolsTable ()

  Gets the current symbols table.
- SymbolsTable ∗ getRootSymbolsTable ()

  Gets the root symbols table.
- int getCurrentOffset ()

  Gets the current offset for variables.
- map< string, CFG ∗ > getCFGS ()

  Gets the control flow graphs for each function.
- map< string, bool > collectSymbolsUsage (SymbolsTable ∗table)

  Collects the usage status of symbols within a given symbols table.

3.5.1 Detailed Description

A visitor class for checking code correctness.

This class extends the `ifccBaseVisitor` and is responsible for checking various aspects of code correctness, including variable declarations, assignments, and expressions, during the parsing phase of the compiler.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 CodeCheckVisitor()

```
CodeCheckVisitor::CodeCheckVisitor ( )
```

Constructs a new instance of the CodeCheckVisitor.

3.5.3 Member Function Documentation

3.5.3.1 collectSymbolsUsage()

```
map< string, bool > CodeCheckVisitor::collectSymbolsUsage (
            SymbolsTable * table )
```

Collects the usage status of symbols within a given symbols table.

This method recursively traverses the symbols table and collects the usage status of each symbol, storing them in a map.

Parameters

table	The symbols table to traverse.

Returns

A map containing the usage status of symbols.

3.5.3.2 getCFGS()

```
map< string, CFG * > CodeCheckVisitor::getCFGS ( )  [inline]
```

Gets the control flow graphs for each function.

Returns

A map containing the control flow graphs indexed by function names.

3.5.3.3 getCurrentOffset()

```
int CodeCheckVisitor::getCurrentOffset ( )  [inline]
```

Gets the current offset for variables.

Returns

The current offset for variables.

3.5.3.4 getCurrentSymbolsTable()

```
SymbolsTable * CodeCheckVisitor::getCurrentSymbolsTable ( )  [inline]
```

Gets the current symbols table.

Returns

The current symbols table.

3.5.3.5 getRootSymbolsTable()

```
SymbolsTable * CodeCheckVisitor::getRootSymbolsTable ( )  [inline]
```

Gets the root symbols table.

Returns

The root symbols table.

3.5.3.6 visitAddsub()

```
antlrcpp::Any CodeCheckVisitor::visitAddsub (
            ifccParser::AddsubContext * ctx )  [override], [virtual]
```

Visits an addition or subtraction expression.

This method processes expressions involving addition or subtraction and checks for correctness in terms of variable usage.

Parameters

ctx	The context for the addition or subtraction expression.

Returns

A result of the visit, typically unused.

3.5.3.7 visitAssign_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitAssign_stmt (
            ifccParser::Assign_stmtContext * ctx )  [override], [virtual]
```

Visits an assignment statement in the parsed code.

This method checks whether variables are assigned values correctly and whether the variables involved are defined.

Parameters

ctx	The context for the assignment statement.

Returns

A result of the visit, typically unused.

3.5.3.8 visitBitwise()

```
antlrcpp::Any CodeCheckVisitor::visitBitwise (
            ifccParser::BitwiseContext * ctx )  [override], [virtual]
```

Visits a bitwise operation expression.

This method processes bitwise operations like AND, OR, XOR, etc., and ensures that all variables in these operations are declared.

Parameters

ctx	The context for the bitwise operation expression.

Returns

A result of the visit, typically unused.

3.5.3.9 visitBlock()

```
antlrcpp::Any CodeCheckVisitor::visitBlock (
            ifccParser::BlockContext * ctx )  [override], [virtual]
```

Visits a block of code in the parsed code.

This method ensures correct variable scoping and handles nested blocks properly.

Parameters

ctx	The context for the block of code.

Returns

A result of the visit, typically unused.

3.5.3.10 visitCall_func_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitCall_func_stmt (
            ifccParser::Call_func_stmtContext * ctx )  [override], [virtual]
```

Visits a function call statement.

This method checks whether the called function exists and matches the expected arguments.

Parameters

ctx	The context for the function call statement.

Returns

A result of the visit, typically unused.

3.5.3.11 visitComp()

```
antlrcpp::Any CodeCheckVisitor::visitComp (
            ifccParser::CompContext * ctx )  [override], [virtual]
```

Visits a comparison expression.

This method processes comparison operations like equality, inequality, greater than, less than, etc., and checks for any correctness issues regarding the variables used.

Parameters

ctx	The context for the comparison expression.

Returns

A result of the visit, typically unused.

3.5.3.12 visitDecl_func_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitDecl_func_stmt (
            ifccParser::Decl_func_stmtContext * ctx )  [override], [virtual]
```

Visits a function declaration statement.

This method records information about the function being declared, such as its parameter count.

Parameters

ctx	The context for the function declaration statement.

Returns

A result of the visit, typically unused.

3.5.3.13 visitDecl_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitDecl_stmt (
            ifccParser::Decl_stmtContext * ctx )  [override], [virtual]
```

Visits a declaration statement in the parsed code.

This method ensures that the variable being declared is correctly handled, ensuring no variable is declared multiple times, or is used before being declared.

Parameters

ctx	The context for the declaration statement.

Returns

A result of the visit, typically unused.

3.5.3.14 visitExpr()

```
antlrcpp::Any CodeCheckVisitor::visitExpr (
            ifccParser::ExprContext * expr )  [virtual]
```

Visits any expression in the parsed code.

This method processes expressions and checks whether variables in expressions are valid and properly declared.

Parameters

expr	The expression context to check.

Returns

A result of the visit, typically unused.

3.5.3.15 visitMuldiv()

```
antlrcpp::Any CodeCheckVisitor::visitMuldiv (
            ifccParser::MuldivContext * ctx )  [override], [virtual]
```

Visits a multiplication or division expression.

This method processes multiplication and division expressions and checks whether variables involved are correctly declared.

Parameters

ctx	The context for the multiplication or division expression.

Returns

A result of the visit, typically unused.

3.5.3.16 visitPost()

```
antlrcpp::Any CodeCheckVisitor::visitPost (
            ifccParser::PostContext * ctx )  [override], [virtual]
```

Visits a post-unary operation (e.g., postfix increment/decrement).

This method processes post-unary operations and checks for correctness in usage.

Parameters

ctx	The context for the post-unary expression.

Returns

A result of the visit, typically unused.

3.5.3.17 visitPost_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitPost_stmt (
            ifccParser::Post_stmtContext * ctx )  [override], [virtual]
```

Visits a post statement (e.g., postfix increment/decrement).

This method processes post statements and checks for correctness in usage.

Parameters

ctx	The context for the post statement.

Returns

A result of the visit, typically unused.

3.5.3.18 visitPre()

```
antlrcpp::Any CodeCheckVisitor::visitPre (
            ifccParser::PreContext * ctx )  [override], [virtual]
```

Visits a pre-unary operation (e.g., prefix increment/decrement).

This method checks for correctness in expressions involving pre-unary operations.

Parameters

ctx	The context for the pre-unary expression.

Returns

A result of the visit, typically unused.

3.5.3.19 visitPre_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitPre_stmt (
            ifccParser::Pre_stmtContext * ctx )  [override], [virtual]
```

Visits a pre statement (e.g., prefix increment/decrement).

This method processes pre-statements and checks for correctness in usage.

Parameters

ctx	The context for the pre-statement.

Returns

A result of the visit, typically unused.

3.5.3.20 visitProg()

```
antlrcpp::Any CodeCheckVisitor::visitProg (
            ifccParser::ProgContext * ctx )  [override], [virtual]
```

Visits the program context in the parsed code.

This method performs a global check of the program, ensuring all variables are declared and used properly.

Parameters

ctx	The context for the program.

Returns

A result of the visit, typically unused.

3.5.3.21 visitReturn_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitReturn_stmt (
            ifccParser::Return_stmtContext * ctx )  [override], [virtual]
```

Visit a return statement in the parsed code.

This method verify the correctness of a return statement.

Parameters

ctx	The context for the return statement.

Returns

A result of the visit, typically unused.

3.5.3.22 visitUnary()

```
antlrcpp::Any CodeCheckVisitor::visitUnary (
            ifccParser::UnaryContext * ctx )  [override], [virtual]
```

Visits a unary expression.

This method processes unary expressions (e.g., negation or logical NOT) and ensures that all variables in the expression are valid.

Parameters

ctx	The context for the unary expression.

Returns

A result of the visit, typically unused.

3.5.3.23 visitVar()

```
antlrcpp::Any CodeCheckVisitor::visitVar (
            ifccParser::VarContext * ctx )  [override], [virtual]
```

Visits a variable context in the parsed code.

This method verifies that the variable is declared and used correctly.

Parameters

ctx	The context for the variable.

Returns

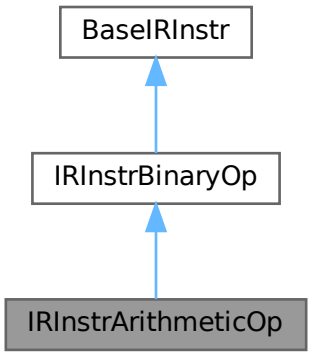
A result of the visit, typically unused.

3.6 IRInstrArithmeticOp Class Reference

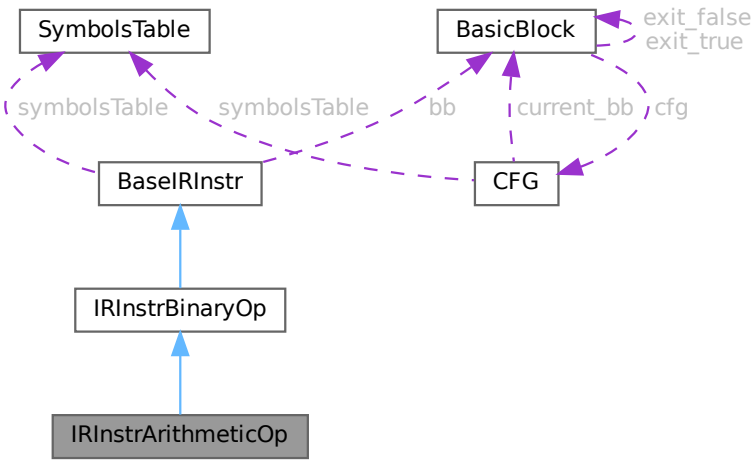
Represents an arithmetic operation instruction in the intermediate representation.

`#include <IRInstrArithmeticOp.h>`

Inheritance diagram for IRInstrArithmeticOp:



Collaboration diagram for IRInstrArithmeticOp:



Public Member Functions

- IRInstrArithmeticOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs an arithmetic operation instruction.
- virtual void genASM (ostream &o)

  Generates the assembly code for this arithmetic operation instruction.

Public Member Functions inherited from IRInstrBinaryOp

- IRInstrBinaryOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a binary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from IRInstrBinaryOp

- string firstOp

  The first operand for the binary operation.
- string secondOp

  The second operand for the binary operation.
- string op

  The binary operation (e.g., '+', '-', '', '/',").*

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.6.1 Detailed Description

Represents an arithmetic operation instruction in the intermediate representation.

This class handles the generation of intermediate representation instructions for arithmetic operations, such as addition, subtraction, multiplication, division, modulo, and bitwise operations. It extends the `IRInstrBinaryOp` class and provides specialized methods for handling these operations.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 IRInstrArithmeticOp()

```
IRInstrArithmeticOp::IRInstrArithmeticOp (
            BasicBlock * bb_,
            string firstOp,
            string secondOp,
            string op )  [inline]
```

Constructs an arithmetic operation instruction.

Initializes the instruction with a basic block, two operands, and the arithmetic operation.

Parameters

*bb_*	The basic block to which the instruction belongs.
firstOp	The first operand of the arithmetic operation.
secondOp	The second operand of the arithmetic operation.
op	The arithmetic operation (e.g., '+', '-', '∗', '/', '%').

3.6.3 Member Function Documentation

3.6.3.1 genASM()

```
void IRInstrArithmeticOp::genASM (
            ostream & o )  [virtual]
```

Generates the assembly code for this arithmetic operation instruction.

This method generates the appropriate assembly code based on the specific arithmetic operation (e.g., addition, subtraction, multiplication, division, modulo, or bitwise operations).

Parameters

o	The output stream where the generated assembly code will be written.

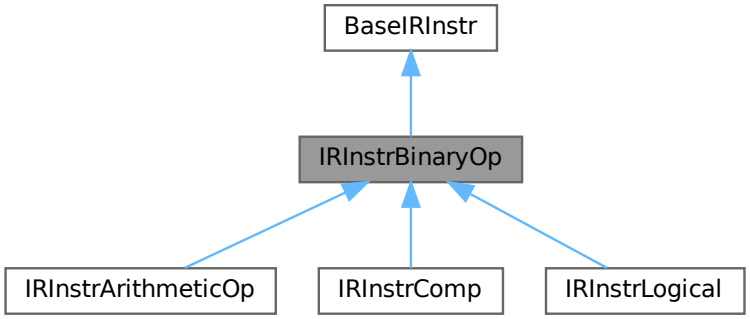
Implements IRInstrBinaryOp.

3.7 IRInstrBinaryOp Class Reference

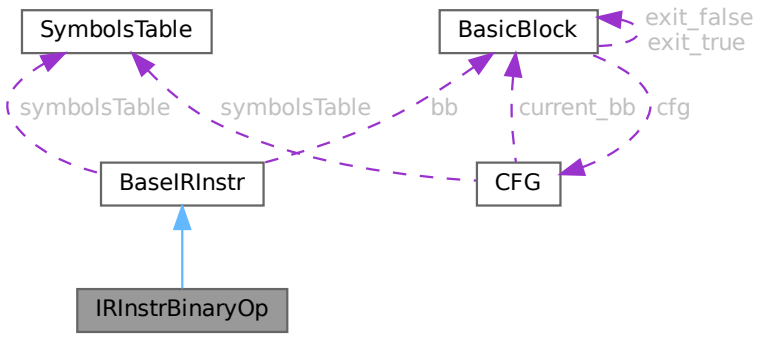
Represents a binary operation instruction in the intermediate representation.

`#include <IRInstrBinaryOp.h>`

Inheritance diagram for IRInstrBinaryOp:



Collaboration diagram for IRInstrBinaryOp:



Public Member Functions

- IRInstrBinaryOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a binary operation instruction.
- virtual void genASM (ostream &o)=0

  Generates the assembly code for this binary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Protected Attributes

- string firstOp

  The first operand for the binary operation.
- string secondOp

  The second operand for the binary operation.
- string op

  The binary operation (e.g., '+', '-', '', '/',").*

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.7.1 Detailed Description

Represents a binary operation instruction in the intermediate representation.

This class serves as the base class for binary operation instructions such as addition, subtraction, multiplication, etc. It provides a structure for managing two operands and the operation itself, and it also handles the generation of the corresponding assembly code for binary operations.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 IRInstrBinaryOp()

```
IRInstrBinaryOp::IRInstrBinaryOp (
            BasicBlock * bb_,
            string firstOp,
            string secondOp,
            string op )  [inline]
```

Constructs a binary operation instruction.

Initializes the instruction with a basic block, two operands, and the binary operation.

Parameters

*bb_*	The basic block to which the instruction belongs.
firstOp	The first operand of the binary operation.
secondOp	The second operand of the binary operation.
op	The binary operation (e.g., '+', '-', '*', '/').

3.7.3 Member Function Documentation

3.7.3.1 genASM()

```
virtual void IRInstrBinaryOp::genASM (
            ostream & o )  [pure virtual]
```

Generates the assembly code for this binary operation instruction.

This method must be implemented by derived classes to generate the appropriate assembly code based on the specific binary operation.

Parameters

o	The output stream where the generated assembly code will be written.

Implements BaseIRInstr.

Implemented in IRInstrArithmeticOp, IRInstrComp, and IRInstrLogical.

3.7.4 Member Data Documentation

3.7.4.1 firstOp

```
string IRInstrBinaryOp::firstOp  [protected]
```

The first operand for the binary operation.

3.7.4.2 op

```
string IRInstrBinaryOp::op  [protected]
```

The binary operation (e.g., '+', '-', '*', '/',").

3.7.4.3 secondOp

```
string IRInstrBinaryOp::secondOp  [protected]
```

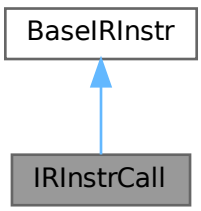
The second operand for the binary operation.

3.8 IRInstrCall Class Reference

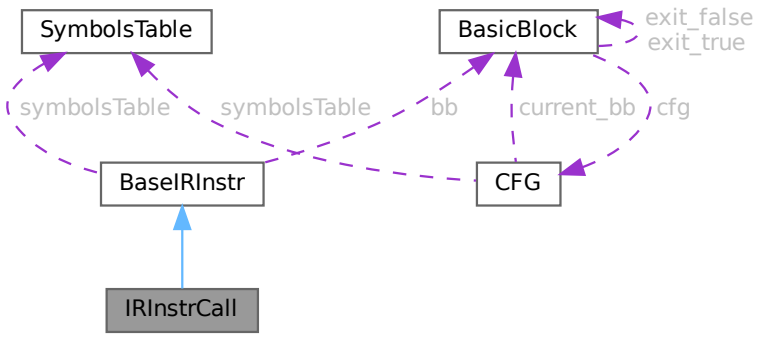
A class representing an intermediate representation instruction for a function call.

```
#include <IRInstrCall.h>
```

Inheritance diagram for IRInstrCall:



Collaboration diagram for IRInstrCall:



Public Member Functions

- IRInstrCall (BasicBlock *bb, const string funcName)

  Constructs an IRInstrCall object.
- void genASM (ostream &o) override

  Generates the assembly code for this function call instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.8.1 Detailed Description

A class representing an intermediate representation instruction for a function call.

This class extends the BaseIRInstr class and represents an instruction for making a function call. It includes information about the basic block it belongs to and the name of the function being called.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 IRInstrCall()

```
IRInstrCall::IRInstrCall (
            BasicBlock * bb,
            const string funcName )  [inline]
```

Constructs an IRInstrCall object.

Parameters

bb	The basic block to which this instruction belongs.
funcName	The name of the function being called.

3.8.3 Member Function Documentation

3.8.3.1 genASM()

```
void IRInstrCall::genASM (
            ostream & o )  [override], [virtual]
```

Generates the assembly code for this function call instruction.

Parameters

o	The output stream where the generated assembly code will be written.

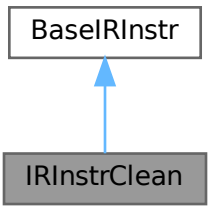
Implements BaseIRInstr.

3.9 IRInstrClean Class Reference

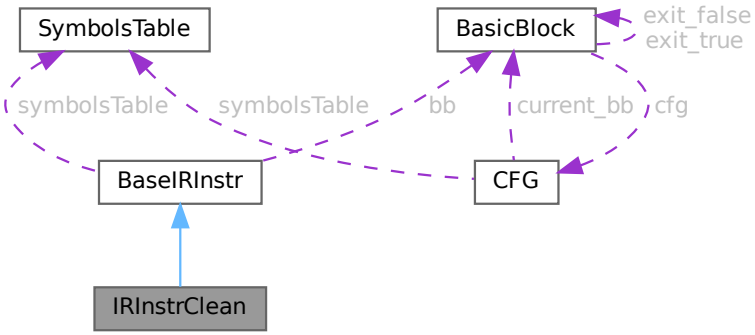
Represents a clean-up instruction in the intermediate representation.

`#include <IRInstrClean.h>`

Inheritance diagram for IRInstrClean:



Collaboration diagram for IRInstrClean:



Public Member Functions

- IRInstrClean (BasicBlock *bb_)

  Constructs an IRInstrClean object.
- virtual void genASM (std::ostream &o) override

  Generates assembly code for the IRInstrClean instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.
- virtual void genASM (ostream &o)=0

  Generates the assembly code for this instruction.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.9.1 Detailed Description

Represents a clean-up instruction in the intermediate representation.

The `IRInstrClean` class is a subclass of `BaseIRInstr`. It represents a clean-up instruction, typically used to deallocate resources or reset values within a basic block during intermediate representation generation. The main responsibility of this class is to generate the corresponding assembly code for the clean-up operation.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 IRInstrClean()

```
IRInstrClean::IRInstrClean (
            BasicBlock * bb_ )  [inline]
```

Constructs an IRInstrClean object.

This constructor initializes an `IRInstrClean` instance with a reference to the basic block where this clean-up instruction resides.

Parameters

*bb_*	A pointer to the `BasicBlock` to which this instruction belongs.

3.9.3 Member Function Documentation

3.9.3.1 genASM()

```
void IRInstrClean::genASM (
            std::ostream & o )  [override], [virtual]
```

Generates assembly code for the IRInstrClean instruction.

This function generates the assembly code corresponding to the clean-up operation represented by this instruction and writes it to the provided output stream.

The generated assembly code typically involves operations to reset, deallocate, or clean up resources associated with the instruction.

Parameters

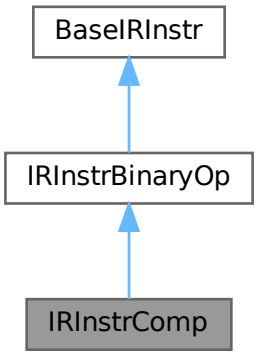
o	The output stream to which the generated assembly code will be written.

3.10 IRInstrComp Class Reference

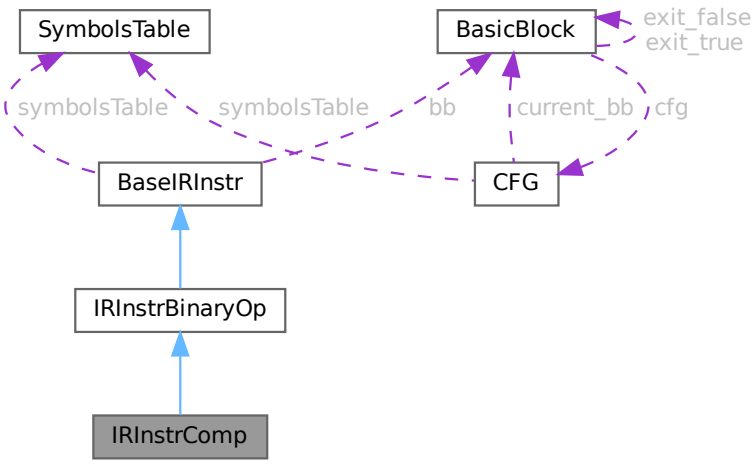
Represents a comparison operation instruction in the intermediate representation.

```
#include <IRInstrComp.h>
```

Inheritance diagram for IRInstrComp:



Collaboration diagram for IRInstrComp:



Public Member Functions

- IRInstrComp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a comparison operation instruction.
- virtual void genASM (ostream &o) override

  Generates the assembly code for this comparison operation instruction.

Public Member Functions inherited from IRInstrBinaryOp

- IRInstrBinaryOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a binary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from IRInstrBinaryOp

- string firstOp

  The first operand for the binary operation.
- string secondOp

  The second operand for the binary operation.
- string op

  The binary operation (e.g., '+', '-', '', '/',").*

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.10.1 Detailed Description

Represents a comparison operation instruction in the intermediate representation.

This class handles the generation of intermediate representation instructions for comparison operations, such as equality, inequality, greater than, greater than or equal to, less than, and less than or equal to.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 IRInstrComp()

```
IRInstrComp::IRInstrComp (
            BasicBlock * bb_,
            string firstOp,
            string secondOp,
            string op )  [inline]
```

Constructs a comparison operation instruction.

Initializes the instruction with a basic block, two operands, and the comparison operation.

Parameters

*bb_*	The basic block to which the instruction belongs.
firstOp	The first operand of the comparison operation.
secondOp	The second operand of the comparison operation.
op	The comparison operation (e.g., '==', '!=', '>', '<', '>=', '<=').

3.10.3 Member Function Documentation

3.10.3.1 genASM()

```
void IRInstrComp::genASM (
            ostream & o )  [override], [virtual]
```

Generates the assembly code for this comparison operation instruction.

This method generates the appropriate assembly code based on the specific comparison operation (e.g., equality, inequality, greater than, greater than or equal to, less than, or less than or equal to).

Parameters

o	The output stream where the generated assembly code will be written.

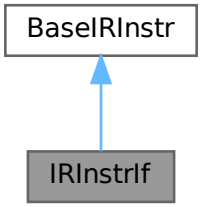
Implements IRInstrBinaryOp.

3.11 IRInstrIf Class Reference

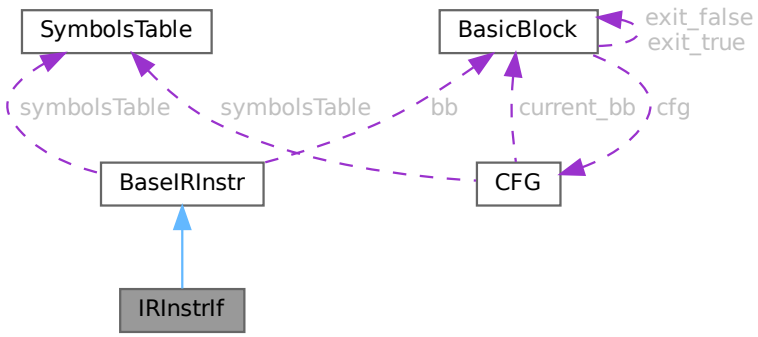
A class representing an if instruction in the intermediate representation.

`#include <IRInstrIf.h>`

Inheritance diagram for IRInstrIf:



Collaboration diagram for IRInstrIf:



Public Member Functions

- IRInstrIf (BasicBlock *bb_, string condReg, string label)

  Constructor for the IRInstrIf class.
- virtual void genASM (ostream &o) override

  Generate assembly code for the if instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.11.1 Detailed Description

A class representing an if instruction in the intermediate representation.

This class handles the creation and generation of assembly code for if statements in the compiler.

Parameters

*bb_*	Pointer to the BasicBlock object associated with this instruction.
condReg	String containing the name of the condition register used in the comparison.
label	String containing the label

3.11.2 Constructor & Destructor Documentation

3.11.2.1 IRInstrIf()

```
IRInstrIf::IRInstrIf (
            BasicBlock * bb_,
            string condReg,
            string label )  [inline]
```

Constructor for the IRInstrIf class.

Creates an instance of the IRInstrIf class with the provided parameters.

Parameters

*bb_*	The basic block to which the instruction belongs.
condReg	Register used in the comparison.
label	Label of the next block

3.11.3 Member Function Documentation

3.11.3.1 genASM()

```
void IRInstrIf::genASM (
            ostream & o )  [override], [virtual]
```

Generate assembly code for the if instruction.

This method generates the appropriate assembly code based on the condition register and label.

Parameters

o	Output stream to write the assembly code.

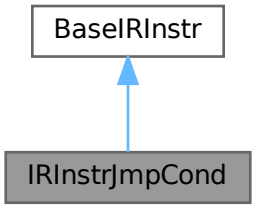
Implements BaseIRInstr.

3.12 IRInstrJmpCond Class Reference

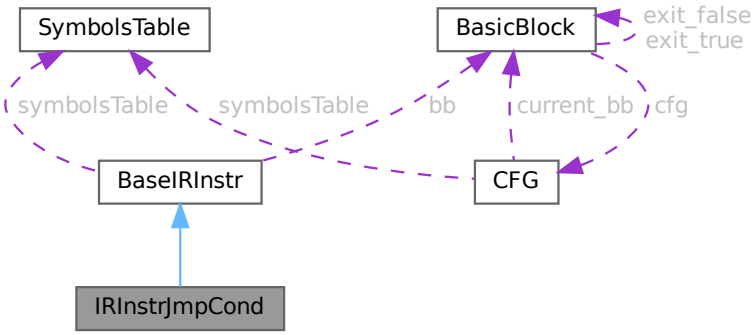
A class representing a conditional jump instruction in the IR.

`#include <IRInstrJmpCond.h>`

Inheritance diagram for IRInstrJmpCond:



Collaboration diagram for IRInstrJmpCond:



Public Member Functions

- IRInstrJmpCond (BasicBlock *bb, string condition, string label, string reg)

  Construct a new IRInstrJmpCond object.
- virtual void genASM (ostream &o) override

  Generate assembly code for the conditional jump instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.12.1 Detailed Description

A class representing a conditional jump instruction in the IR.

This class extends the BaseIRInstr class and represents a conditional jump instruction.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 IRInstrJmpCond()

```
IRInstrJmpCond::IRInstrJmpCond (
            BasicBlock * bb,
            string condition,
            string label,
            string reg )  [inline]
```

Construct a new IRInstrJmpCond object.

Parameters

bb	The basic block to which this instruction belongs.
condition	The condition under which the jump occurs.
label	The destination label for the jump.
reg	The register to use for testing the condition.

3.12.3 Member Function Documentation

3.12.3.1 genASM()

```
void IRInstrJmpCond::genASM (
            ostream & o )  [override], [virtual]
```

Generate assembly code for the conditional jump instruction.

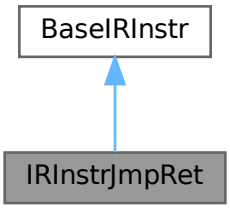
Parameters

o	The output stream to write the generated assembly code.

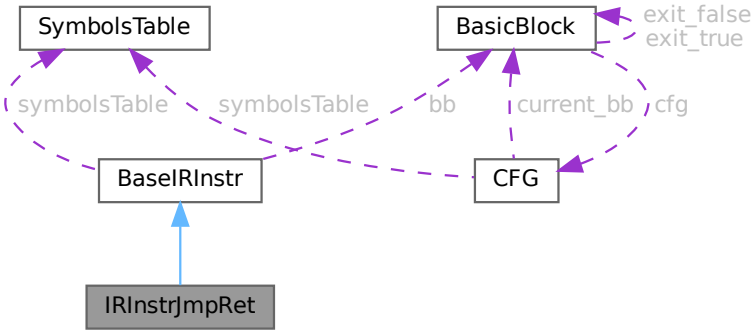
Implements BaseIRInstr.

3.13 IRInstrJmpRet Class Reference

Inheritance diagram for IRInstrJmpRet:



Collaboration diagram for IRInstrJmpRet:



Public Member Functions

- IRInstrJmpRet (BasicBlock *bb_, string label)
  Constructs an IRInstrJmpRet object.
- virtual void genASM (ostream &o) override
  Generates the assembly code for ret instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)
  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()
  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.13.1 Constructor & Destructor Documentation

3.13.1.1 IRInstrJmpRet()

```
IRInstrJmpRet::IRInstrJmpRet (
            BasicBlock * bb_,
            string label )  [inline]
```

Constructs an IRInstrJmpRet object.

Parameters

*bb_*	The basic block to which this instruction belongs.
label	The label of the basic block to jump to after returning.

3.13.2 Member Function Documentation

3.13.2.1 genASM()

```
void IRInstrJmpRet::genASM (
            ostream & o )  [override], [virtual]
```

Generates the assembly code for ret instruction.

Parameters

o	The output stream where the generated assembly code will be written.

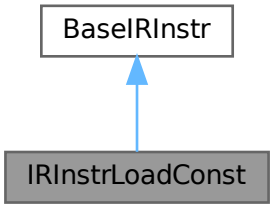
Implements BaseIRInstr.

3.14 IRInstrLoadConst Class Reference

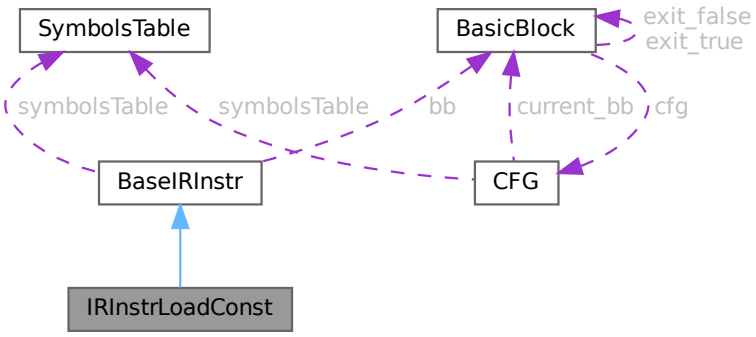
Represents an IR instruction for loading a constant into memory or a register.

```
#include <IRInstrLoadConst.h>
```

Inheritance diagram for IRInstrLoadConst:



Collaboration diagram for IRInstrLoadConst:



Public Member Functions

- IRInstrLoadConst (BasicBlock *bb_, int value, string dest)

  Constructor for the IRInstrLoadConst instruction.
- virtual void genASM (ostream &o) override

  Generates assembly code to load a constant.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.14.1 Detailed Description

Represents an IR instruction for loading a constant into memory or a register.

This instruction is used to assign an immediate value to a register or memory location.

3.14.2 Constructor & Destructor Documentation

3.14.2.1 IRInstrLoadConst()

```
IRInstrLoadConst::IRInstrLoadConst (
            BasicBlock * bb_,
            int value,
            string dest )
```

Constructor for the IRInstrLoadConst instruction.

Initializes the instruction with the basic block, the constant value to load, and the destination register or memory variable where the value should be stored.

Parameters

*bb_*	Pointer to the basic block containing this instruction.
value	The constant value to load.
dest	The name of the target register or memory variable.

3.14.3 Member Function Documentation

3.14.3.1 genASM()

```
void IRInstrLoadConst::genASM (
            ostream & o )  [override], [virtual]
```

Generates assembly code to load a constant.

Generates assembly code to load a constant into memory or a register.

Generates assembly code for loading a constant value into a register or memory.

Parameters

o	Output stream where the assembly code will be written.

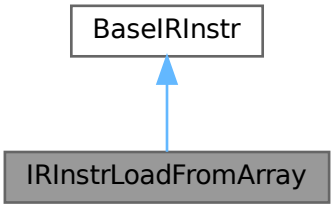
Implements BaseIRInstr.

3.15 IRInstrLoadFromArray Class Reference

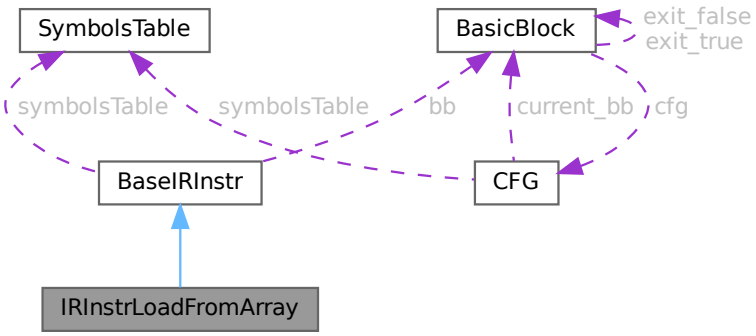
A class representing an instruction that loads an array element into a register.

`#include <IRInstrLoadFromArray.h>`

Inheritance diagram for IRInstrLoadFromArray:



Collaboration diagram for IRInstrLoadFromArray:



Public Member Functions

- IRInstrLoadFromArray (BasicBlock *bb, string src, string dest, int index)

  Construct a new IRInstrLoadFromArray object.
- virtual void genASM (ostream &o)

  Generate assembly code for loading an array element into a register.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.15.1 Detailed Description

A class representing an instruction that loads an array element into a register.

This class represents an instruction that loads an array element into a register.

3.15.2 Constructor & Destructor Documentation

3.15.2.1 IRInstrLoadFromArray()

```
IRInstrLoadFromArray::IRInstrLoadFromArray (
            BasicBlock * bb,
            string src,
            string dest,
            int index )
```

Construct a new IRInstrLoadFromArray object.

Parameters

bb	The basic block to which this instruction belongs.
src	The source array name.
dest	The destination register.

3.15.3 Member Function Documentation

3.15.3.1 genASM()

```
void IRInstrLoadFromArray::genASM (
            ostream & o )  [virtual]
```

Generate assembly code for loading an array element into a register.

Parameters

o	The output stream to write the generated assembly code.

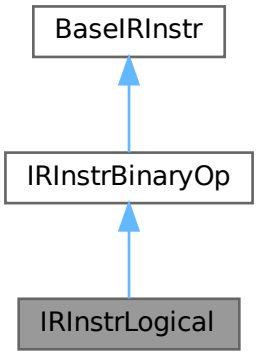
Implements BaseIRInstr.

3.16 IRInstrLogical Class Reference

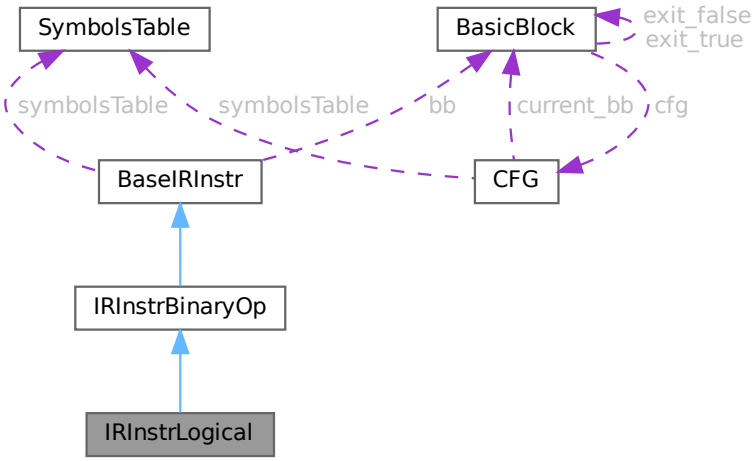
A class representing a logical operation instruction.

`#include <IRInstrLogical.h>`

Inheritance diagram for IRInstrLogical:



Collaboration diagram for IRInstrLogical:



Public Member Functions

- IRInstrLogical (BasicBlock *bb_, string firstOp, string secondOp, string op)
  Constructs a logical operation instruction.
- virtual void genASM (ostream &o) override
  Generates the assembly code for this logical operation instruction.

Public Member Functions inherited from IRInstrBinaryOp

- IRInstrBinaryOp (BasicBlock *bb_, string firstOp, string secondOp, string op)
  Constructs a binary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)
  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()
  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from IRInstrBinaryOp

- string firstOp
  The first operand for the binary operation.
- string secondOp
  The second operand for the binary operation.
- string op
  The binary operation (e.g., '+', '-', '', '/',").*

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb
  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable
  The symbol table for the current scope.

3.16.1 Detailed Description

A class representing a logical operation instruction.

This class represents an instruction for the lazy evaluation version of the logical AND (&&) and OR (||)

3.16.2 Constructor & Destructor Documentation

3.16.2.1 IRInstrLogical()

```
IRInstrLogical::IRInstrLogical (
            BasicBlock * bb_,
            string firstOp,
            string secondOp,
            string op )  [inline]
```

Constructs a logical operation instruction.

Initializes the instruction with a basic block, two operands, and the logical operation.

Parameters

*bb_*	The basic block to which the instruction belongs.
firstOp	The first operand of the logical operation.
secondOp	The second operand of the logical operation.
op	The logical operation (e.g., '&&', '||').

3.16.3 Member Function Documentation

3.16.3.1 genASM()

```
void IRInstrLogical::genASM (
            ostream & o )  [override], [virtual]
```

Generates the assembly code for this logical operation instruction.

Parameters

o	The output stream where the generated assembly code will be written.

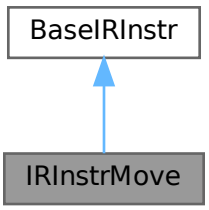
Implements IRInstrBinaryOp.

3.17 IRInstrMove Class Reference

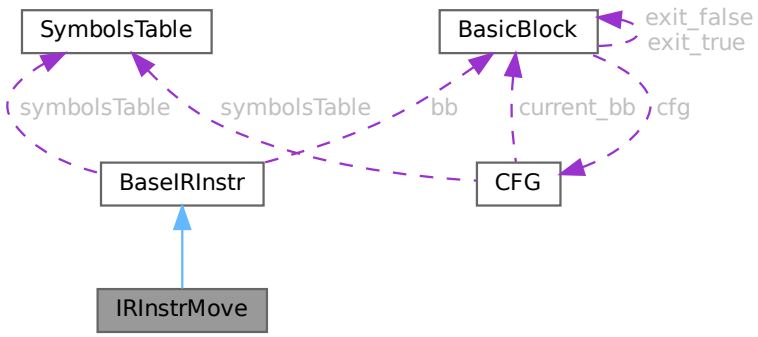
Represents an IR instruction for moving a value between registers and memory.

```
#include <IRInstrMove.h>
```

Inheritance diagram for IRInstrMove:



Collaboration diagram for IRInstrMove:



Public Member Functions

- IRInstrMove (BasicBlock *bb_, string src, string dest)

  Constructor for the IRInstrMove instruction.
- virtual void genASM (ostream &o) override

  Generates the assembly code corresponding to the move instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.17.1 Detailed Description

Represents an IR instruction for moving a value between registers and memory.

This class handles data transfers between registers and the stack but does not manage constants.

3.17.2 Constructor & Destructor Documentation

3.17.2.1 IRInstrMove()

```
IRInstrMove::IRInstrMove (
            BasicBlock * bb_,
            string src,
            string dest )
```

Constructor for the IRInstrMove instruction.

Initializes the instruction with the basic block, source, and destination variables.

Parameters

*bb_*	Pointer to the basic block containing this instruction.
src	The name of the source variable (register or memory location).
dest	The name of the destination variable (register or memory location).

3.17.3 Member Function Documentation

3.17.3.1 genASM()

```
void IRInstrMove::genASM (
            ostream & o )  [override], [virtual]
```

Generates the assembly code corresponding to the move instruction.

Generates assembly code for moving a value between registers and memory.

This method generates the appropriate assembly code for moving a value from the source variable to the destination variable.

Parameters

o	Output stream where the assembly code will be written.

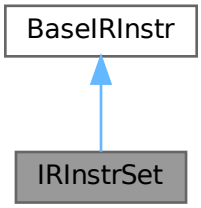
Implements BaseIRInstr.

3.18 IRInstrSet Class Reference

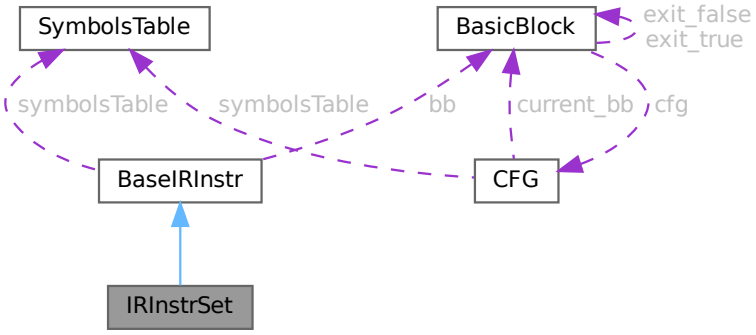
Represents an instruction that sets a value in the intermediate representation.

```
#include <IRInstrSet.h>
```

Inheritance diagram for IRInstrSet:



Collaboration diagram for IRInstrSet:



Public Member Functions

- IRInstrSet (BasicBlock ∗bb_)

  Constructs an IRInstrSet object.
- virtual void genASM (std::ostream &o) override

  Generates assembly code for the IRInstrSet instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock ∗bb_)

  Constructs an instruction for a given basic block.
- BasicBlock ∗ getBB ()

  Gets the basic block that this instruction belongs to.
- virtual void genASM (ostream &o)=0

  Generates the assembly code for this instruction.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.18.1 Detailed Description

Represents an instruction that sets a value in the intermediate representation.

The `IRInstrSet` class is a subclass of `BaseIRInstr`. It represents an instruction that performs an assignment or setting of a value within a basic block during intermediate representation generation. The primary function of this class is to generate the assembly code for the instruction.

3.18.2 Constructor & Destructor Documentation

3.18.2.1 IRInstrSet()

```
IRInstrSet::IRInstrSet (
            BasicBlock * bb_ )  [inline]
```

Constructs an IRInstrSet object.

This constructor initializes an `IRInstrSet` instance with a reference to the basic block in which this instruction resides.

Parameters

*bb_*	A pointer to the `BasicBlock` to which this instruction belongs.

3.18.3 Member Function Documentation

3.18.3.1 genASM()

```
void IRInstrSet::genASM (
            std::ostream & o )  [override], [virtual]
```

Generates assembly code for the IRInstrSet instruction.

This function generates the corresponding assembly code for the `IRInstrSet` instruction and writes it to the provided output stream.

The generated assembly code typically includes an instruction for setting a value in a register or memory location, depending on the context.

Parameters

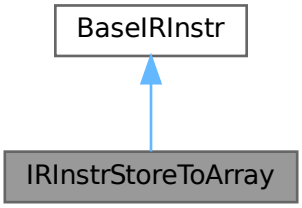
o	The output stream to which the generated assembly code will be written.

3.19 IRInstrStoreToArray Class Reference

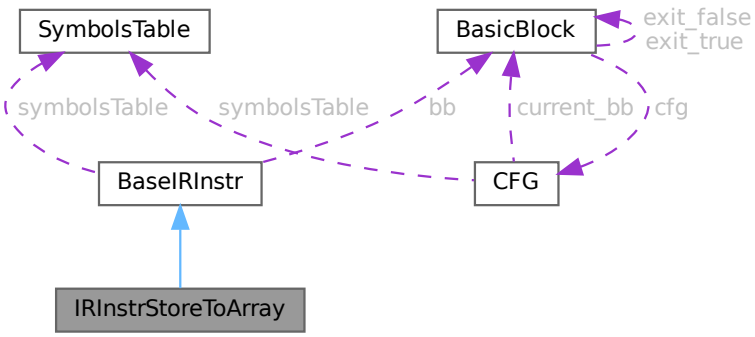
A class representing an instruction to store data at an array location.

`#include <IRInstrStoreToArray.h>`

Inheritance diagram for IRInstrStoreToArray:



Collaboration diagram for IRInstrStoreToArray:



Public Member Functions

- IRInstrStoreToArray (BasicBlock *bb, int baseOffset, string indexRegister, string sourceRegister)

  Construct a new IRInstrStoreToArray object.
- virtual void genASM (ostream &o)

  Generate assembly code for storing data at an array location.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.19.1 Detailed Description

A class representing an instruction to store data at an array location.

3.19.2 Constructor & Destructor Documentation

3.19.2.1 IRInstrStoreToArray()

```
IRInstrStoreToArray::IRInstrStoreToArray (
            BasicBlock * bb,
            int baseOffset,
            string indexRegister,
            string sourceRegister )
```

Construct a new IRInstrStoreToArray object.

Parameters

bb	The basic block to which this instruction belongs.
baseOffset	The offset from the start of the array.
indexRegister	The register holding the index of the array element to store.
sourceRegister	The register holding the value to be stored.

3.19.3 Member Function Documentation

3.19.3.1 genASM()

```
void IRInstrStoreToArray::genASM (
            ostream & o )  [virtual]
```

Generate assembly code for storing data at an array location.

Parameters

o	The output stream to write the generated assembly code.

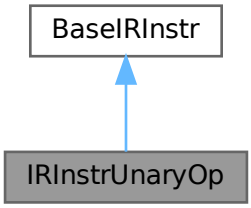
Implements BaseIRInstr.

3.20 IRInstrUnaryOp Class Reference

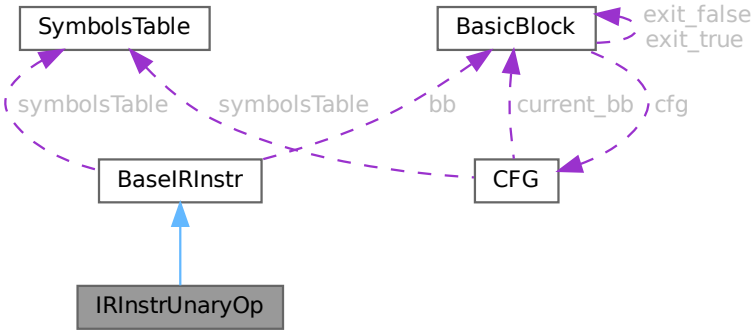
Represents a unary operation instruction in the intermediate representation.

`#include <IRInstrUnaryOp.h>`

Inheritance diagram for IRInstrUnaryOp:



Collaboration diagram for IRInstrUnaryOp:



Public Member Functions

- IRInstrUnaryOp (BasicBlock *bb_, string uniqueOp, string op)

  Constructs a unary operation instruction.
- virtual void genASM (ostream &o)

  Generates the assembly code for this unary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Protected Attributes

- string uniqueOp

  A unique identifier for the unary operation.
- string op

  The actual unary operation (e.g., '!', '-', '~').

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

3.20.1 Detailed Description

Represents a unary operation instruction in the intermediate representation.

This class handles the generation of intermediate representation instructions for unary operations, such as negation, logical negation, and bitwise complement.

3.20.2 Constructor & Destructor Documentation

3.20.2.1 IRInstrUnaryOp()

```
IRInstrUnaryOp::IRInstrUnaryOp (
            BasicBlock * bb_,
            string uniqueOp,
            string op )  [inline]
```

Constructs a unary operation instruction.

Initializes the instruction with a basic block, a unique operation identifier, and the operation itself.

Parameters

*bb_*	The basic block to which the instruction belongs.
uniqueOp	A unique identifier for the operation.
op	The actual unary operation (e.g., '!', '-', '~').

3.20.3 Member Function Documentation

3.20.3.1 genASM()

```
void IRInstrUnaryOp::genASM (
            ostream & o )  [virtual]
```

Generates the assembly code for this unary operation instruction.

This method generates the appropriate assembly code based on the specific unary operation (e.g., negation, logical NOT, bitwise complement).

Parameters

o	The output stream where the generated assembly code will be written.

Implements BaseIRInstr.

3.20.4 Member Data Documentation

3.20.4.1 op

```
string IRInstrUnaryOp::op  [protected]
```

The actual unary operation (e.g., '!', '-', '~').

3.20.4.2 uniqueOp

```
string IRInstrUnaryOp::uniqueOp  [protected]
```

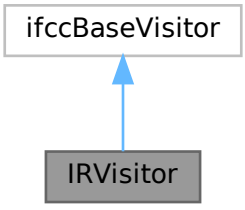
A unique identifier for the unary operation.

3.21 IRVisitor Class Reference

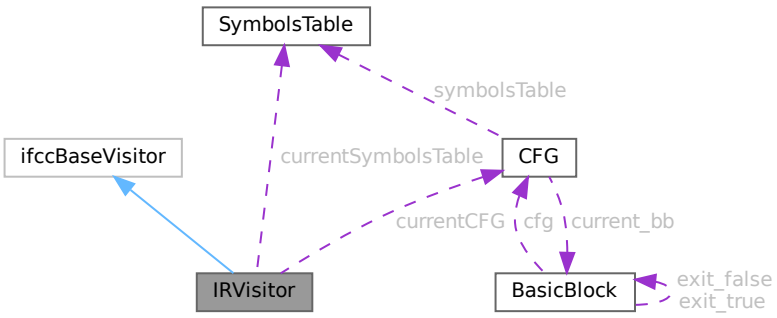
A visitor class for generating Intermediate Representation (IR) during parsing.

`#include <IRVisitor.h>`

Inheritance diagram for IRVisitor:



Collaboration diagram for IRVisitor:



Public Member Functions

- IRVisitor (map< string, CFG * > cfgs)

  Constructs an IRVisitor.
- virtual antlrcpp::Any visitProg (ifccParser::ProgContext *ctx) override

  Visits the program and starts the IR generation process.
- virtual antlrcpp::Any visitBlock (ifccParser::BlockContext *ctx) override

  Visits a block of code and generates the IR.
- virtual antlrcpp::Any visitReturn_stmt (ifccParser::Return_stmtContext *ctx) override

  Visits a return statement and generates the IR.
- virtual antlrcpp::Any visitAssign_stmt (ifccParser::Assign_stmtContext *ctx) override

  Visits an assignment statement and generates the IR.
- virtual antlrcpp::Any visitAssign (ifccParser::AssignContext *ctx) override

  Visits an assignment expression in the parse tree.
- virtual antlrcpp::Any visitDecl_stmt (ifccParser::Decl_stmtContext *ctx) override

  Visits a declaration statement and generates the IR.
- virtual antlrcpp::Any visitArray_access (ifccParser::Array_accessContext *ctx) override

  Visits an array access expression and generates the IR.
- antlrcpp::Any visitExpr (ifccParser::ExprContext *expr, string targetRegister)

  Visits an expression and generates the IR.
- virtual antlrcpp::Any visitAddsub (ifccParser::AddsubContext *ctx) override

  Visits an addition or subtraction expression and generates the IR.
- virtual antlrcpp::Any visitMuldiv (ifccParser::MuldivContext *ctx) override

  Visits a multiplication or division expression and generates the IR.
- virtual antlrcpp::Any visitBitwise (ifccParser::BitwiseContext *ctx) override

  Visits a bitwise operation expression and generates the IR.
- virtual antlrcpp::Any visitComp (ifccParser::CompContext *ctx) override

  Visits a comparison expression and generates the IR.
- virtual antlrcpp::Any visitUnary (ifccParser::UnaryContext *ctx) override

  Visits a unary expression and generates the IR.
- virtual antlrcpp::Any visitShift (ifccParser::ShiftContext *ctx) override

  Visits a shift operation (e.g., left or right shift) and generates the IR.
- virtual antlrcpp::Any visitLogicalAND (ifccParser::LogicalANDContext *ctx) override

  Visits a logical AND expression and generates the IR.

- virtual antlrcpp::Any visitLogicalOR (ifccParser::LogicalORContext ∗ctx) override

  Visits a logical OR expression and generates the IR.
- void genASM (ostream &o)

  Generates the assembly code for the IR.
- virtual antlrcpp::Any visitPre (ifccParser::PreContext ∗ctx) override

  Visits a pre-unary operation (e.g., prefix increment/decrement) and generates the IR.
- virtual antlrcpp::Any visitPre_stmt (ifccParser::Pre_stmtContext ∗ctx) override

  Visits a pre-statement (e.g., prefix increment/decrement) and generates the IR.
- virtual antlrcpp::Any visitPost (ifccParser::PostContext ∗ctx) override

  Visits a post-unary operation (e.g., postfix increment/decrement) and generates the IR.
- virtual antlrcpp::Any visitPost_stmt (ifccParser::Post_stmtContext ∗ctx) override

  Visits a post-statement (e.g., postfix increment/decrement) and generates the IR.
- virtual antlrcpp::Any visitDecl_func_stmt (ifccParser::Decl_func_stmtContext ∗ctx) override

  Visits a call function statement and generates the IR.
- virtual antlrcpp::Any visitCall_func_stmt (ifccParser::Call_func_stmtContext ∗ctx) override

  Visits a call function statement and generates the IR.
- virtual antlrcpp::Any visitIf_stmt (ifccParser::If_stmtContext ∗ctx) override

  Visits an if statement and generates the IR.
- virtual antlrcpp::Any visitWhile_stmt (ifccParser::While_stmtContext ∗ctx) override

  Visits a while statement and generates the IR.
- map< string, CFG ∗ > getCFGS ()

  Retrieves the map of Control Flow Graphs (CFGs).
- SymbolsTable ∗ getCurrentSymbolsTable ()

  Retrieves the current symbols table. This method retrieves the current symbols table that is being used for the IR generation.
- void setCurrentSymbolsTable (SymbolsTable ∗currentSymbolsTable)

  Sets the current symbols table. This method sets the current symbols table that is being used for the IR generation.

Protected Attributes

- map< string, CFG ∗ > cfgs

  A map of variable names to their corresponding Control Flow Graphs (CFGs).
- map< SymbolsTable ∗, int > childIndices

  A map of symbols tables to their corresponding indices in theirs scope.
- CFG ∗ currentCFG

  The current control flow graph (CFG) being used.
- SymbolsTable ∗ currentSymbolsTable

  The current symbols table being used.

3.21.1 Detailed Description

A visitor class for generating Intermediate Representation (IR) during parsing.

This class extends the `ifccBaseVisitor` and is responsible for traversing the parsed code to generate the Intermediate Representation (IR), such as the Control Flow Graph (CFG), for the code being compiled.

3.21.2 Constructor & Destructor Documentation

3.21.2.1 IRVisitor()

```
IRVisitor::IRVisitor (
            map< string, CFG * > cfgs )
```

Constructs an IRVisitor.

Initializes the IRVisitor with a symbols table and base stack offset.

Parameters

symbolsTable	A map containing variable names and their associated stack offsets.
baseStackOffset	The base offset for the stack.

3.21.3 Member Function Documentation

3.21.3.1 genASM()

```
void IRVisitor::genASM (
            ostream & o )
```

Generates the assembly code for the IR.

This method generates assembly code from the Intermediate Representation (IR) for output.

Parameters

o	The output stream where the assembly code will be written.

3.21.3.2 getCFGS()

```
map< string, CFG * > IRVisitor::getCFGS ( )
```

Retrieves the map of Control Flow Graphs (CFGs).

This function returns a map where the keys are string labels (e.g., function names or block labels) and the values are pointers to the corresponding `CFG` objects. These `CFG` objects represent the control flow graphs of different sections of the parsed program.

The returned map can be used for further analysis, visualization (e.g., by generating Graphviz `.dot` files), or manipulation of the program's control flow.

Returns

A `std::map` where the key is a string label representing the section of the program (such as a function name) and the value is a pointer to the corresponding `CFG` object.

3.21.3.3 getCurrentSymbolsTable()

```
SymbolsTable * IRVisitor::getCurrentSymbolsTable ( )  [inline]
```

Retrieves the current symbols table. This method retrieves the current symbols table that is being used for the IR generation.

Returns

The current symbols table.

3.21.3.4 setCurrentSymbolsTable()

```
void IRVisitor::setCurrentSymbolsTable (
            SymbolsTable * currentSymbolsTable )
```

Sets the current symbols table. This method sets the current symbols table that is being used for the IR generation.

Parameters

currentSymbolsTable	A pointer to the current symbols table.

3.21.3.5 visitAddsub()

```
antlrcpp::Any IRVisitor::visitAddsub (
            ifccParser::AddsubContext * ctx )  [override], [virtual]
```

Visits an addition or subtraction expression and generates the IR.

This method processes addition and subtraction operations and generates the corresponding IR.

Parameters

ctx	The context of the addition or subtraction expression.

Returns

A result of the visit, typically unused.

3.21.3.6 visitArray_access()

```
antlrcpp::Any IRVisitor::visitArray_access (
            ifccParser::Array_accessContext * ctx )  [override], [virtual]
```

Visits an array access expression and generates the IR.

This method processes array access expressions and generates the corresponding IR for accessing elements in arrays.

Parameters

ctx	The context of the array access expression.

Returns

A result of the visit, typically unused.

3.21.3.7 visitAssign()

```
antlrcpp::Any IRVisitor::visitAssign (
            ifccParser::AssignContext * ctx )  [override], [virtual]
```

Visits an assignment expression in the parse tree.

This method processes an assignment (=) in the input C code. It retrieves the variable being assigned, evaluates the right-hand side expression, and generates the necessary Intermediate Representation (IR) instructions.

Parameters

ctx	The context of the assignment node in the parse tree.

Returns

The result of processing the assignment, wrapped in `antlrcpp::Any`.

3.21.3.8 visitAssign_stmt()

```
antlrcpp::Any IRVisitor::visitAssign_stmt (
            ifccParser::Assign_stmtContext * ctx )  [override], [virtual]
```

Visits an assignment statement and generates the IR.

This method processes assignment statements and generates the corresponding IR for the variable assignment.

Parameters

ctx	The context of the assignment statement.

Returns

A result of the visit, typically unused.

3.21.3.9 visitBitwise()

```
antlrcpp::Any IRVisitor::visitBitwise (
            ifccParser::BitwiseContext * ctx )  [override], [virtual]
```

Visits a bitwise operation expression and generates the IR.

This method processes bitwise operations like AND, OR, XOR, etc., and generates the corresponding IR.

Parameters

ctx	The context of the bitwise operation expression.

Returns

A result of the visit, typically unused.

3.21.3.10 visitBlock()

```
antlrcpp::Any IRVisitor::visitBlock (
            ifccParser::BlockContext * ctx )  [override], [virtual]
```

Visits a block of code and generates the IR.

This method processes blocks of code and generates the corresponding IR for each statement within the block.

Parameters

ctx	The context of the block.

Returns

A result of the visit, typically unused.

3.21.3.11 visitCall_func_stmt()

```
antlrcpp::Any IRVisitor::visitCall_func_stmt (
            ifccParser::Call_func_stmtContext * ctx )  [override], [virtual]
```

Visits a call function statement and generates the IR.

This method processes calls to functions and generates the corresponding IR for calling those functions.

Parameters

ctx	The context of the call function statement.

Returns

A result of the visit, typically unused.

3.21.3.12 visitComp()

```
antlrcpp::Any IRVisitor::visitComp (
            ifccParser::CompContext * ctx )  [override], [virtual]
```

Visits a comparison expression and generates the IR.

This method processes comparison operations (e.g., equality, greater-than) and generates the corresponding IR.

Parameters

ctx	The context of the comparison expression.

Returns

A result of the visit, typically unused.

3.21.3.13 visitDecl_func_stmt()

```
antlrcpp::Any IRVisitor::visitDecl_func_stmt (
            ifccParser::Decl_func_stmtContext * ctx )  [override], [virtual]
```

Visits a call function statement and generates the IR.

This method processes calls to functions and generates the corresponding IR for calling those functions.

Parameters

ctx	The context of the call function statement.

Returns

A result of the visit, typically unused.

3.21.3.14 visitDecl_stmt()

```
antlrcpp::Any IRVisitor::visitDecl_stmt (
            ifccParser::Decl_stmtContext * ctx )  [override], [virtual]
```

Visits a declaration statement and generates the IR.

This method processes declaration statements and generates the corresponding IR for the variable declaration.

Parameters

ctx	The context of the declaration statement.

Returns

A result of the visit, typically unused.

3.21.3.15 visitExpr()

```
antlrcpp::Any IRVisitor::visitExpr (
            ifccParser::ExprContext * expr,
            string targetRegister )
```

Visits an expression and generates the IR.

This method processes expressions and generates the corresponding IR for the expression.

Parameters

expr	The expression context to generate IR for.
targetRegister	The register to store the result of the expression evaluation.

Returns

A result of the visit, typically unused.

3.21.3.16 visitIf_stmt()

```
antlrcpp::Any IRVisitor::visitIf_stmt (
            ifccParser::If_stmtContext * ctx )  [override], [virtual]
```

Visits an if statement and generates the IR.

This method processes if statements and generates the corresponding IR for the conditional branching.

Parameters

ctx	The context of the if statement.

Returns

A result of the visit, typically unused.

3.21.3.17 visitLogicalAND()

```
antlrcpp::Any IRVisitor::visitLogicalAND (
            ifccParser::LogicalANDContext * ctx )  [override], [virtual]
```

Visits a logical AND expression and generates the IR.

This method processes logical AND operations and generates the corresponding IR.

Parameters

ctx	The context of the logical AND expression.

Returns

A result of the visit, typically unused.

3.21.3.18 visitLogicalOR()

```
antlrcpp::Any IRVisitor::visitLogicalOR (
            ifccParser::LogicalORContext * ctx )  [override], [virtual]
```

Visits a logical OR expression and generates the IR.

This method processes logical OR operations and generates the corresponding IR.

Parameters

ctx	The context of the logical OR expression.

Returns

A result of the visit, typically unused.

3.21.3.19 visitMuldiv()

```
antlrcpp::Any IRVisitor::visitMuldiv (
            ifccParser::MuldivContext * ctx )  [override], [virtual]
```

Visits a multiplication or division expression and generates the IR.

This method processes multiplication and division operations and generates the corresponding IR.

Parameters

ctx	The context of the multiplication or division expression.

Returns

A result of the visit, typically unused.

3.21.3.20 visitPost()

```
antlrcpp::Any IRVisitor::visitPost (
            ifccParser::PostContext * ctx )  [override], [virtual]
```

Visits a post-unary operation (e.g., postfix increment/decrement) and generates the IR.

This method processes post-unary operations and generates the corresponding IR.

Parameters

ctx	The context of the post-unary expression.

Returns

A result of the visit, typically unused.

3.21.3.21 visitPost_stmt()

```
antlrcpp::Any IRVisitor::visitPost_stmt (
            ifccParser::Post_stmtContext * ctx )  [override], [virtual]
```

Visits a post-statement (e.g., postfix increment/decrement) and generates the IR.

This method processes post-statements and generates the corresponding IR.

Parameters

ctx	The context of the post-statement.

Returns

A result of the visit, typically unused.

3.21.3.22 visitPre()

```
antlrcpp::Any IRVisitor::visitPre (
            ifccParser::PreContext * ctx )  [override], [virtual]
```

Visits a pre-unary operation (e.g., prefix increment/decrement) and generates the IR.

This method processes pre-unary operations and generates the corresponding IR.

Parameters

ctx	The context of the pre-unary expression.

Returns

A result of the visit, typically unused.

3.21.3.23 visitPre_stmt()

```
antlrcpp::Any IRVisitor::visitPre_stmt (
            ifccParser::Pre_stmtContext * ctx )  [override], [virtual]
```

Visits a pre-statement (e.g., prefix increment/decrement) and generates the IR.

This method processes pre-statements and generates the corresponding IR.

Parameters

ctx	The context of the pre-statement.

Returns

A result of the visit, typically unused.

3.21.3.24 visitProg()

```
antlrcpp::Any IRVisitor::visitProg (
            ifccParser::ProgContext * ctx )  [override], [virtual]
```

Visits the program and starts the IR generation process.

This method starts the process of visiting the program node, generating the IR for the entire program.

Parameters

ctx	The context of the program.

Returns

A result of the visit, typically unused.

3.21.3.25 visitReturn_stmt()

```
antlrcpp::Any IRVisitor::visitReturn_stmt (
            ifccParser::Return_stmtContext * ctx )  [override], [virtual]
```

Visits a return statement and generates the IR.

This method processes return statements and generates the corresponding IR for the return operation.

Parameters

ctx	The context of the return statement.

Returns

A result of the visit, typically unused.

3.21.3.26 visitShift()

```
antlrcpp::Any IRVisitor::visitShift (
            ifccParser::ShiftContext * ctx )  [override], [virtual]
```

Visits a shift operation (e.g., left or right shift) and generates the IR.

This method processes shift operations and generates the corresponding IR.

Parameters

ctx	The context of the shift operation.

Returns

A result of the visit, typically unused.

3.21.3.27 visitUnary()

```
antlrcpp::Any IRVisitor::visitUnary (
            ifccParser::UnaryContext * ctx )  [override], [virtual]
```

Visits a unary expression and generates the IR.

This method processes unary operations (e.g., negation or logical NOT) and generates the corresponding IR.

Parameters

ctx	The context of the unary expression.

Returns

A result of the visit, typically unused.

3.21.3.28 visitWhile_stmt()

```
antlrcpp::Any IRVisitor::visitWhile_stmt (
            ifccParser::While_stmtContext * ctx )  [override], [virtual]
```

Visits a while statement and generates the IR.

This method processes while statements and generates the corresponding IR for the loop.

Parameters

ctx	The context of the while statement.

Returns

A result of the visit, typically unused.

3.21.4 Member Data Documentation

3.21.4.1 cfgs

```
map<string, CFG *> IRVisitor::cfgs  [protected]
```

A map of variable names to their corresponding Control Flow Graphs (CFGs).

3.21.4.2 childIndices

```
map<SymbolsTable *, int> IRVisitor::childIndices  [protected]
```

A map of symbols tables to their corresponding indices in theirs scope.

3.21.4.3 currentCFG

```
CFG* IRVisitor::currentCFG  [protected]
```

The current control flow graph (CFG) being used.

3.21.4.4 currentSymbolsTable

```
SymbolsTable* IRVisitor::currentSymbolsTable  [protected]
```

The current symbols table being used.

3.22 SymbolsTable Class Reference

Stores information about variables,including their names, types, usage status, and indexes.

```
#include <SymbolsTable.h>
```

Public Member Functions

- SymbolsTable (int currentOffset=-4)
  Construct a new Symbols Table object with an optional initial offset.
- void addSymbol (string name, Type type, int symbolSize, int index=0)
  Add the symbol to the current symbol table and update the offset.
- void addChild (SymbolsTable *child)
  Add a child symbol table to the current one.
- int getSymbolIndex (string name)
  Get the symbol index.
- Type getSymbolType (string name)
  Get the symbol type.
- bool symbolIsUsed (string name)
  Returns true if the symbol is used.
- bool symbolHasAValue (string name)
  Returns true if the symbol has been assigned a value.
- void setSymbolUsage (string name, bool isUsed)
  Sets the usage status of a symbol.
- map< string, bool > getSymbolsUsage ()
  Returns the symbols usage mapping variable names to their usage status.
- void setSymbolDefinitionStatus (string name, bool hasValue)
  Sets the definition status of a symbol (whether it has been assigned a value).
- bool containsSymbol (string name)
  Returns true if the symbol table contains the symbol.
- map< string, int > getSymbolsIndex ()
  Returns the symbols index mapping variable names to their offsets.
- map< string, Type > getSymbolsType ()
  Returns the symbols type mapping variable names to their types.
- map< string, bool > getSymbolsDefinitionStatus ()
  Returns the symbols definition status mapping variable names to their assignment status.
- vector< SymbolsTable * > getChildren ()
  Returns the child symbol tables.
- SymbolsTable * getParent ()
  Returns the parent symbol table.
- void **setParent** (SymbolsTable *parent)

3.22.1 Detailed Description

Stores information about variables,including their names, types, usage status, and indexes.

3.22.2 Constructor & Destructor Documentation

3.22.2.1 SymbolsTable()

```
SymbolsTable::SymbolsTable (
            int currentOffset = -4 )  [inline]
```

Construct a new Symbols Table object with an optional initial offset.

Parameters

currentOffset	The initial offset (default is -4).

3.22.3 Member Function Documentation

3.22.3.1 addChild()

```
void SymbolsTable::addChild (
            SymbolsTable * child )
```

Add a child symbol table to the current one.

Parameters

child	The child symbol table to be added.

3.22.3.2 addSymbol()

```
void SymbolsTable::addSymbol (
            string name,
            Type type,
            int symbolSize,
            int index = 0 )
```

Add the symbol to the current symbol table and update the offset.

Parameters

name	The name of the symbol.
type	The type of the symbol.

3.22.3.3 containsSymbol()

```
bool SymbolsTable::containsSymbol (
            string name )
```

Returns true if the symbol table contains the symbol.

Parameters

name	The name of the symbol.

Returns

bool - Whether the symbol exists in the table.

3.22.3.4 getChildren()

```
vector< SymbolsTable * > SymbolsTable::getChildren ( )  [inline]
```

Returns the child symbol tables.

Returns

vector<SymbolsTable *> - The list of child symbol tables.

3.22.3.5 getParent()

```
SymbolsTable * SymbolsTable::getParent ( )  [inline]
```

Returns the parent symbol table.

Returns

SymbolsTable * - The parent symbol table.

3.22.3.6 getSymbolIndex()

```
int SymbolsTable::getSymbolIndex (
            string name )
```

Get the symbol index.

Parameters

name	The name of the symbol.

Returns

int - The index of the symbol.

3.22.3.7 getSymbolsDefinitionStatus()

```
map< string, bool > SymbolsTable::getSymbolsDefinitionStatus ( )  [inline]
```

Returns the symbols definition status mapping variable names to their assignment status.

Returns

map<string, bool> - The symbols definition status.

3.22.3.8 getSymbolsIndex()

```
map< string, int > SymbolsTable::getSymbolsIndex ( )  [inline]
```

Returns the symbols index mapping variable names to their offsets.

Returns

map<string, int> - The symbols index.

3.22.3.9 getSymbolsType()

```
map< string, Type > SymbolsTable::getSymbolsType ( )  [inline]
```

Returns the symbols type mapping variable names to their types.

Returns

map<string, Type> - The symbols type.

3.22.3.10 getSymbolsUsage()

```
map< string, bool > SymbolsTable::getSymbolsUsage ( )  [inline]
```

Returns the symbols usage mapping variable names to their usage status.

Returns

map<string, bool> - The symbols usage.

3.22.3.11 getSymbolType()

```
Type SymbolsTable::getSymbolType (
            string name )
```

Get the symbol type.

Parameters

name	The name of the symbol.

Returns

Type - The type of the symbol.

3.22.3.12 setSymbolDefinitionStatus()

```
void SymbolsTable::setSymbolDefinitionStatus (
            string name,
            bool hasValue )
```

Sets the definition status of a symbol (whether it has been assigned a value).

Parameters

name	The name of the symbol.
hasValue	Whether the symbol has a value assigned.

3.22.3.13 setSymbolUsage()

```
void SymbolsTable::setSymbolUsage (
            string name,
            bool isUsed )
```

Sets the usage status of a symbol.

Parameters

name	The name of the symbol.
isUsed	Whether the symbol has been used.

3.22.3.14 symbolHasAValue()

```
bool SymbolsTable::symbolHasAValue (
            string name )
```

Returns true if the symbol has been assigned a value.

Parameters

name	The name of the symbol.

Returns

bool - Whether the symbol has a value.

3.22.3.15 symbolIsUsed()

```
bool SymbolsTable::symbolIsUsed (
            string name )
```

Returns true if the symbol is used.

Parameters

name	The name of the symbol.

Returns

bool - Whether the symbol has been used.

3.23 TypeManager Class Reference

A class responsible for managing data types.

```
#include <TypeManager.h>
```

Static Public Member Functions

- static Type stringToType (const string &str)

  Converts a string to the corresponding Type enum value.
- static int size_of (Type t)

  Returns the size of a given type.

3.23.1 Detailed Description

A class responsible for managing data types.

3.23.2 Member Function Documentation

3.23.2.1 size_of()

```
int TypeManager::size_of (
            Type t )  [static]
```

Returns the size of a given type.

Parameters

t	The type value.

Returns

The size of the type in bytes.

3.23.2.2 stringToType()

```
Type TypeManager::stringToType (
            const string & str )  [static]
```

Converts a string to the corresponding Type enum value.

Parameters

str	The string representation of the type.

Returns

The corresponding Type enum value.

Index