1. The Symbol Table

The symbol table keeps track of the values, types, functions, and variables of the code, defined by the programmer and well as temporary variables introduced by the compiler. It also holds struct and union tags (enumerations do not have tags). In fact, there are actually several symbol tables. There are symbol tables for the global space, for each function, for each block in the functions, and for the members of each struct or union. In this way, the symbol tables form a hierarchy where each table holds a reference to its parent table (except the table for the global space, which table parent reference is null).

For example, let us look at the following code.

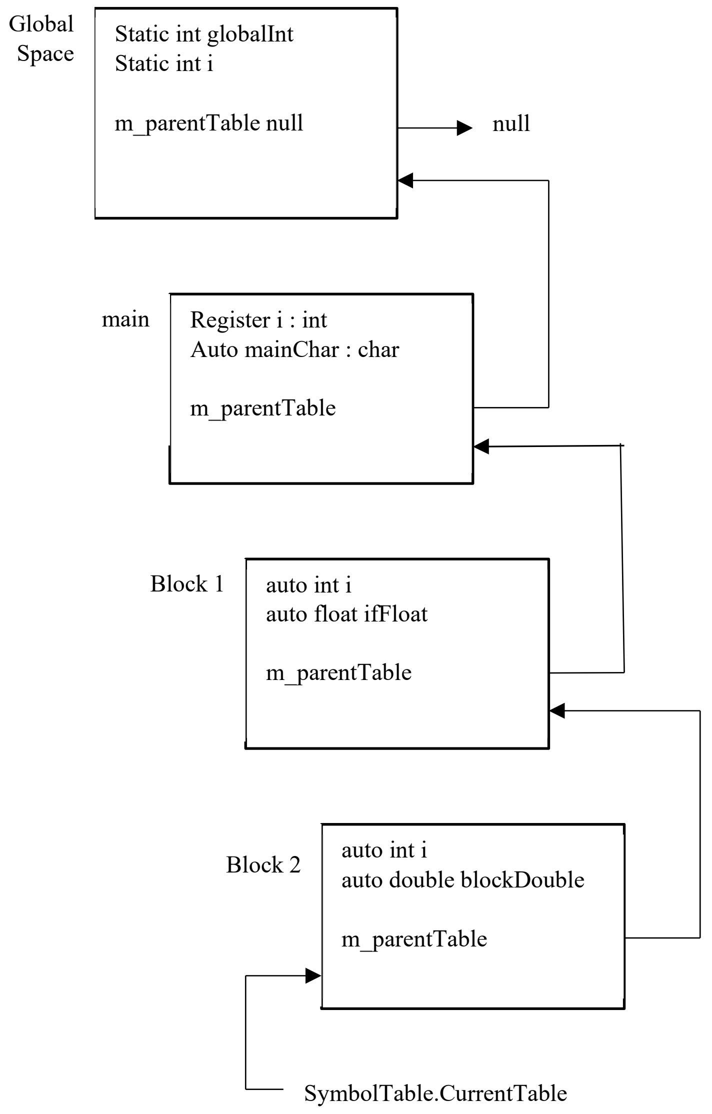
```
int globalInt;
int i;

void main() {
  register int i;
  char mainChar;

  if (globalCount > 0) { // Block 1
    int i;
    float ifFloat;

    { int i; // Block 2
      double blockDouble;
      // Point p
    }
  }
}
```

When the parsing reaches point **p**, the symbol table has the form below. Note that the variable **i** defined in every block and is present in every table. When a symbol is looked up, we start searching in the current table and continue to search up through the hierarchy until we found a symbol with the given name or the parent symbol reference is null, in which case the table representing the global space.



The **SymbolTable** class holds a symbol table. The scope of a table is **Global**, **Function**, **Block**, **Struct**, or **Union** as of the **Scope** enumeration.

SymbolTable.cs
```
using System;
using System.Collections.Generic;

namespace CCompiler {
  public class SymbolTable {
    private Scope m_scope;
```

For each function call, an activation record is allocated at the call stack. The first three entries are the return jump address (the address of the instruction following the function call), the regular frame pointer (the address of the current activation record), and the variadic frame pointer (the regular frame pointer plus the size of potential extra parameters in a variadic call). The function header size is the sum of the size of those three entries. As the compiler in this book can be set to generate code for different target machines, the offsets are also given different values as the pointer size varies.

```
    public static int ReturnAddressOffset = 0;
    public static int RegularFrameOffset = TypeSize.PointerSize;
    public static int VariadicFrameOffset = 2 * TypeSize.PointerSize;
    public static int FunctionHeaderSize = 3 * TypeSize.PointerSize;
```

Each table holds a reference to its parent table, except the table of global space, which table parent reference is null.

```
    private SymbolTable m_parentTable;
```

The activation record is organized in the following way:

- The offsets of the return jump address, the regular frame pointer, and the variadic frame pointer.
- The regular function parameters.
- The extra parameters in case of a variadic call.
- Variables of auto or register storage.
- Temporary integral values are stored are loaded from the registers and stored to the activation record during function calls.
- Temporary floating-point values are loaded from the floating-point stack and to the activation record during function calls.

```
    private int m_currentOffset;
```

All parameters and variables of a function or a block, as well as the members of a struct or a union, are stored in the **m_entryMap** and **m_entryList**. We use **m_entryMap** to look up symbols and **m_entryList** when initializing values in a struct or union, in which case the symbols need to be stored in the order they were declared in the code. The standard C# **Dictionary** class provides fast searching but does not guarantee any order among its key-value pairs.

```
    private IDictionary<string,Symbol> m_entryMap =
      new Dictionary<string,Symbol>();
    private List<Symbol> m_entryList = new List<Symbol>();
```

The tag map holds the types of the structs or unions that are marked with a tag.

```
    private IDictionary<string,Type> m_tagMap =
      new Dictionary<string,Type>();
```

CurrentTable and **CurrentFunction** hold references to the current symbol table and the function currently processed by the parser. We need the current symbol table when adding or looking up symbols, and the current function when generating code for calling a function or returning a value. In global scope, the **CurrentFunction** is null.

```
    public static SymbolTable CurrentTable = null;
    public static Symbol CurrentFunction = null;
```

GlobalStaticSet holds all static symbols. Only symbols of auto or register storage are given offsets. Symbols of static, extern, or typedef storage are not given offsets, since they are not stored at the activation record in runtime.

```
    public static ISet<StaticSymbol> GlobalStaticSet;
    public static StaticSymbolLinux InitSymbol, ArgsSymbol;
```

The constructor of the **SymbolTable** class takes the parent table (which is null for global space) and the scope of the table.

```
    public SymbolTable(SymbolTable parentTable, Scope scope) {
      m_parentTable = parentTable;
```

In case of global scope, we create the global static set. There is only one table of global scope and one static set in each source code file. In this case we do not need to set the offset field since there are no variables of auto or register storage in global scope.

```
      switch (m_scope = scope) {
        case Scope.Global:
          GlobalStaticSet = new HashSet<StaticSymbol>();
          InitSymbol = ArgsSymbol = null;
          break;
```

In case of struct of union scope, the current offset is initialized to zero.

```
        case Scope.Struct:
        case Scope.Union:
          m_currentOffset = 0;
          break;
```

In case of function scope, the current offset is initialized to the size of the header size. The return address, regular frame pointer, and variadic frame pointer holds the first entries, and the parameters are given the offsets following the initialization.

```
        case Scope.Function:
          m_currentOffset = FunctionHeaderSize;
          break;
```

In case of block scope, the current offset is initialized to the current offset of its parent table. When the parsing leaves a block the parent table again becomes the current table, and a following block is again initialized to the current offset of the parent table. In this way, symbols of different blocks can share the same space on the activation record.

```
        case Scope.Block:
          m_currentOffset = m_parentTable.m_currentOffset;
          break;
      }
    }

    public Scope Scope {
      get { return m_scope; }
    }

    public SymbolTable ParentTable {
      get { return m_parentTable; }
    }

    public IDictionary<string,Symbol> EntryMap {
      get { return m_entryMap; }
    }

    public List<Symbol> EntryList {
      get { return m_entryList; }
    }

    public int CurrentOffset {
      get { return m_currentOffset; }
    }
```

The **AddSymbol** method adds a symbol to the symbol table. It is possible to add two symbols with the same name if they have equal types and at least one of them holds extern storage. When adding a symbol with a name we have to check a few features. If the symbol is extern, it cannot be initialized (its value must be null).

```
    public void AddSymbol(Symbol newSymbol) {
      string name = newSymbol.Name;
```

The name of the symbol may be null in case of unnamed function parameters.

```
      if (name != null) {
        Symbol oldSymbol;
```

We look up the entry list, and if we find another symbol with the same name, we check whether at least one of the symbols is extern and they have equals types. Otherwise, we have two non-extern symbols with the same name, and we report an error.

```
        if (m_entryMap.TryGetValue(name, out oldSymbol)) {
          Assert.Error(oldSymbol.IsExtern() || newSymbol.IsExtern(),
                       name, Message.Name_already_defined);
          Assert.Error(oldSymbol.Type.Equals(newSymbol.Type),
                       name, Message.Different_types_in_redeclaration);
```

The new symbol is given the same unique name as the old symbol, in order for the linker to find the symbol. Since the symbols are referred to by the same name, they also need to have the same unique name,

```
          newSymbol.UniqueName = oldSymbol.UniqueName;
```

If the new symbol is not extern, we remove the old symbol and replace it with the new symbol. We do not need to modify **m_entryList**, since we use int in struct and unions only, and struct or union members cannot be extern.

```
          if (!newSymbol.IsExtern()) {
            m_entryMap[name] = newSymbol;
          }
        }
```

If there is no previous symbol with the same name, we just add the new symbol to the entry map and entry list.

```
        else {
          m_entryMap[name] = newSymbol;
          m_entryList.Add(newSymbol);
        }
      }
```

If the variable is auto or register and not a function, it shall be given an offset in the activation record. Unless in union scope, in which each member always has offset zero.

```
      if (!newSymbol.Type.IsFunction()) {
        if (newSymbol.IsAutoOrRegister()) {
          if (m_scope == Scope.Union) {
            newSymbol.Offset = 0;
          }
```

If the new symbol is an enumeration constant item, is shall not be given an offset. This is because we do not know at this point if the item holds auto or register scope. An item is given auto storage from the beginning, but that may change when the whole enumeration is given its storage, which is done after the item is stored in the symbol table.

```
          else if (!newSymbol.Type.IsEnumerator()) {
            newSymbol.Offset = m_currentOffset;
            m_currentOffset += newSymbol.Type.Size();
          }
        }
      }
    }
```

The **SetOffset** method sets the offset of a symbol. It is called when an enumeration constant item has finally been given auto or register storage.

```
    public void SetOffset(Symbol symbol) {
      symbol.Offset = m_currentOffset;
      m_currentOffset += symbol.Type.Size();
    }
```

The **LookupSymbol** looks up the symbol with the given in the current symbol table and its parent tables, recursively. If no symbol with the given name is found, null is returned. Note that this method searches the parent tables while **AddSymbol** only checks for symbols with the same name in the current table.

```
public Symbol LookupSymbol(string name) {
  Symbol symbol;

  if (m_entryMap.TryGetValue(name, out symbol)) {
    return symbol;
  }
  else if (m_parentTable != null) {
    return m_parentTable.LookupSymbol(name);
  }

  return null;
}
```

The **AddTag** method adds a struct or union tag to the current symbol table. It is possible to add a new tag with an already taken name. However, in that case, the old and the new tag must have the same sort (they must both be structs or both be unions) and the member map of at least one of them must be null. This order is necessary to provide linked lists in C.

```
public void AddTag(string name, Type newType) {
  if (m_tagMap.ContainsKey(name)) {
    Type oldType = m_tagMap[name];
    Assert.Error(!oldType.IsEnumerator() &&
                 (oldType.Sort == newType.Sort), name,
                 Message.Name_already_defined);
```

If the member map of the old tag is null, we assign it the member map of the new tag.

```
    if (oldType.MemberMap == null) {
      oldType.MemberMap = newType.MemberMap;
      oldType.MemberList = newType.MemberList;
    }
```

If the neither the old nor the new member map is null, we report an error since it is not allowed to have two structs or unions with the same name tags with non-null member maps.

```
    else {
      Assert.Error(newType.MemberMap == null, name,
                   Message.Name_already_defined);
    }
  }
```

If there is no struct or union tag previous added to the tag map, we simply add the new tag to the map.

```
  else {
    m_tagMap.Add(name, newType);
  }
}
```

The **LookUpTag** method looks up a tag in the current table or its parent tables. If the tag is not found, null is returned. Similar to **LookUpSymbol**, **LookUpTag** searches the parent tables.

```
public Type LookupTag(string name, Sort sort) {
  Type type;

  if (m_tagMap.TryGetValue(name, out type) && (type.Sort == sort)) {
    return type;
  }
  else if (m_parentTable != null) {
        return m_parentTable.LookupTag(name, sort);
      }

      return null;
    }
  }
}
```