# 1.1. The Linker

The linker is the final part of the compilation process. Its merges together the compiled files and generates an executable file.

## 1.1.1. The Linker Class

The **Linker** class merge code of the object files and modifies the global accesses, function calls, and return assignments, and saves the final code in a .com file, which is executable in 16-bits Windows. More specific, the linker has the following tasks:

- First, we load all function and static variables from all the object files and to make sure that two elements do not share the same name.
- Then we identify the **main** function and trace all functions and global variables reachable from **main**. All elements not reachable from **main** are removed.
- For each function or global variable, we modify all accesses, and for each function we modify all calls and return assignments.
- Finally, we save the code and data for all elements reachable from **main** in the .com file.

**Linker.cs**

```
using System;
using System.IO;
using System.Collections.Generic;

namespace CCompiler {
  public class Linker {
```

There are several maps and lists. The **m_globalMap** map holds all the static symbols of the source code, while **m_globalList** holds only the symbols that are reachable (directly or indirectly) from the **main** function. The final code is generated from the symbols of **m_globalList**, the other symbols are omitted from the final code. The **m_addressMap** holds the address of each symbol in **m_globalList**. Finally, **m_totalSize** holds the current total size of the symbols of **m_globalList** and is used to add the positions of the symbols to **m_addressMap**. The **StackStart** field is used to identify the first address after the code and data.

```
    private IDictionary<string,StaticSymbolWindows> m_globalMap =
      new Dictionary<string,StaticSymbolWindows>();
    private List<StaticSymbolWindows> m_globalList =
      new List<StaticSymbolWindows>();
    private IDictionary<string,int> m_addressMap =
      new Dictionary<string,int>();
    private int m_totalSize = 256;
    public static string StackStart = Symbol.SeparatorId + "StackTop";
```

The **Add** method is called for each static symbol of the source code. If two symbols have the same unique name not ending in the number identification character (‘#’), we have two symbols with external linkage, which is not allowed. However, if the name ends with the identification character, it only means that two numerical constants with the same value is present in two different files, with is allowed. In that case we simply refrain from adding the second symbol to the map.

```
    public void Add(StaticSymbol staticSymbol) {
      StaticSymbolWindows staticSymbolWindows =
        (StaticSymbolWindows) staticSymbol;
      string uniqueName = staticSymbolWindows.UniqueName;

      if (!m_globalMap.ContainsKey(uniqueName)) {
        m_globalMap.Add(uniqueName, staticSymbolWindows);
```

```
    }
    else {
      Assert.Error(uniqueName.EndsWith(Symbol.NumberId),
                   SimpleName(uniqueName),
                   Message.Duplicate_global_name);
    }
  }
```

The **Generate** method writes the final executable code to the target file. Its task is to identify the symbols reachable from the **main** function, and to replace the names of accessed symbols with called functions with proper addresses as well as to replace the function returns addresses with proper addresses. However, we need to start by including the code for the initialization execution code and, optionally, the code for handling the command line arguments.

```
  public void Generate(FileInfo targetFile) {
```

The code for initialization shall be added to the code at the beginning, before the **main** function

```
    StaticSymbolWindows initializerInfo =
      m_globalMap[AssemblyCodeGenerator.InitializerName];
    m_globalList.Add(initializerInfo);
    m_totalSize += initializerInfo.ByteList.Count;
    m_addressMap.Add(AssemblyCodeGenerator.InitializerName, 0);
```

In case of command line arguments, we add its symbol after the initialization code symbol and before the **main** function code.

```
    StaticSymbolWindows pathNameSymbol = null;
    if (m_globalMap.ContainsKey(AssemblyCodeGenerator.ArgsName)) {
      StaticSymbolWindows argsInfo =
        m_globalMap[AssemblyCodeGenerator.ArgsName];
      m_globalList.Add(argsInfo);
      Console.Out.WriteLine(argsInfo.UniqueName);
      m_totalSize += argsInfo.ByteList.Count;
      m_addressMap.Add(AssemblyCodeGenerator.ArgsName, 0);
```

We also need to add the path name of final executable file.

```
      List<byte> byteList = new List<byte>();
      IDictionary<int, string> accessMap = new Dictionary<int, string>();
      pathNameSymbol = (StaticSymbolWindows)
        ConstantExpression.Value(AssemblyCodeGenerator.PathName,
                                 Type.StringType, @"C:\D\Main.com");
      m_globalMap.Add(AssemblyCodeGenerator.PathName, pathNameSymbol);
    }
```

We look up the main function symbol and report an error if it is not present. If it is present, we add it after the initiation symbol and potential command line symbol. Then we call **GenerateTrace**, which traces all symbols reachable from the main function, and adds them to **m_globalList** and **m_addressMap**.

```
    StaticSymbolWindows mainInfo;
    Assert.Error(m_globalMap.TryGetValue("main", out mainInfo),
                 "non-static main", Message.Function_missing);
    GenerateTrace(mainInfo);
```

In case of command line arguments, we add the symbol for the executable file path name to **m_globalList** and **m_addressMap**.

```
    if (pathNameSymbol != null) {
      Assert.Error(!m_globalList.Contains(pathNameSymbol));
      m_globalList.Add(pathNameSymbol);
      m_addressMap.Add(pathNameSymbol.UniqueName, m_totalSize);
      m_totalSize += (int) pathNameSymbol.ByteList.Count;
```

```
    }
```

Finally, we add the stack top name, which will be the address of the activation record of the **main** function.

```
    m_addressMap.Add(StackStart, m_totalSize);
```

We iterate through the global list and, for each of its symbols, we call **GenerateAccess**, **GenerateCall**, **GenerateReturn**, which replace the names of accessed static symbols and called functions as well as return addresses with proper addresses.

```
    foreach (StaticSymbolWindows staticSymbol in m_globalList) {
      List<byte> byteList = staticSymbol.ByteList;
      int startAddress = m_addressMap[staticSymbol.UniqueName];
      GenerateAccess(staticSymbol.AccessMap, byteList);
      GenerateCall(startAddress, staticSymbol.CallMap, byteList);
      GenerateReturn(startAddress, staticSymbol.ReturnSet, byteList);
    }
```

Finally, we write the byte list of each symbol in the global list to the target file.

```
    { Console.Out.WriteLine("Generating \"" +
                            targetFile.FullName + "\".");
      targetFile.Delete();
      BinaryWriter targetStream =
        new BinaryWriter(File.OpenWrite(targetFile.FullName));

      foreach (StaticSymbolWindows staticSymbol in m_globalList) {
        foreach (sbyte b in staticSymbol.ByteList) {
          targetStream.Write(b);
        }
      }

      targetStream.Close();
    }
  }
```

The **GenerateTrace** method adds the symbol to the **m_globalList** iterates through the names of access name set and function call name set and calls **GenerateTrace** recursively for each symbol not yet present in the **m_globalList**.

```
  private void GenerateTrace(StaticSymbolWindows staticSymbol) {
    if (!m_globalList.Contains(staticSymbol)) {
      m_globalList.Add(staticSymbol);
      m_addressMap.Add(staticSymbol.UniqueName, m_totalSize);
      m_totalSize += (int) staticSymbol.ByteList.Count;

      ISet<string> accessNameSet =
        new HashSet<string>(staticSymbol.AccessMap.Values);
      foreach (string accessName in accessNameSet) {
        StaticSymbolWindows accessSymbol;
        Assert.Error(m_globalMap.TryGetValue(accessName,
                                             out accessSymbol),
                     accessName, Message.Object_missing_in_linking);
        Assert.Error(accessSymbol != null);
        GenerateTrace(accessSymbol);
      }

      ISet<string> callNameSet =
        new HashSet<string>(staticSymbol.CallMap.Values);
      foreach (string callName in callNameSet) {
        StaticSymbolWindows funcSymbol;
        Assert.Error(m_globalMap.TryGetValue(callName, out funcSymbol),
```

```
                    callName, Message.Function_missing_in_linking);
        Assert.Error(funcSymbol != null, SimpleName(callName),
                     Message.Missing_external_function);
        GenerateTrace(funcSymbol);
      }
    }
  }
```

The **GenerateAccess** methods iterators through the access map and modifies the addresses. For each element, reachable from **main**, we need to modify its static accesses, function calls and return assignments (for a global variable the call map and return set are empty) by calling **GenerateAccess**, **GenerateCall**, and **GenerateReturn**.

```
  private void GenerateAccess(IDictionary<int,string> accessMap,
                              List<byte> byteList) {
    foreach (KeyValuePair<int,string> entry in accessMap) {
      int sourceAddress = entry.Key;
      string name = entry.Value;
```

We obtain the target address from the low and high byte of the source address. Note that the target address does not need to be zero, it may hold an offset. For instance, the target address may be a pointer to a value in an array of a non-zero index, in which case its offset is non-zero.

```
      byte lowByte = byteList[sourceAddress],
           highByte = byteList[sourceAddress + 1];
      int targetAddress = ((int) highByte << 8) + lowByte;
```

We modify the target address by adding the address of the name to target address, which we restore to the source address.

```
      targetAddress += m_addressMap[name];
      byteList[sourceAddress] = (byte) targetAddress;
      byteList[sourceAddress + 1] = (byte) (targetAddress >> 8);
    }
  }
```

The **GenerateCall** method iterates through the names of the functions to called. There are two differences between this method and **GenerateAccess** above. First, we do not need to take inspect the original address of the function. It is always zero since it is not possible to add an offset to a function call. Second, the address of the call is not absolute, it is relative the next assembly code instruction.

```
  private void GenerateCall(int startAddress,
                            IDictionary<int, string> callMap,
                            List<byte> byteList) {
    const byte NopOperator = -112 + 256;
    const byte ShortJumpOperator = -21 + 256;

    foreach (KeyValuePair<int,string> entry in callMap) {
      int address = entry.Key;
```

The caller address is the last two bytes of the function assembly code instruction, why the address of the next instruction is the source address plus two. The callee address we simply look up in the **m_addressMap** map. The relative address of the call is the difference between the addresses of the callee and caller function.

```
      int callerAddress = startAddress + address + 2;
      int calleeAddress = m_addressMap[entry.Value];
      int relativeAddress = calleeAddress - callerAddress;
```

If the relative address is more or equals to -128, we change the call to a short jump.

```
      if (relativeAddress >= -128) {
        byteList[address - 1] = (byte) NopOperator;
```

```
          byteList[address] = (byte) ShortJumpOperator;
          byteList[address + 1] = (byte) relativeAddress;
        }
```

If the relative address is exactly -129, we have a special case. We change the call to a short jump, but move the call one byte backwards.

```
        else if (relativeAddress == -129) {
          byteList[address - 1] = (byte) ShortJumpOperator;
          byteList[address] = (byte) (-128 + 256);
          byteList[address + 1] = (byte) NopOperator;
        }
```

Otherwise, we just set the address of the call.

```
        else {
          byteList[address] = (byte) ((sbyte) relativeAddress);
          byteList[address + 1] = (byte) ((sbyte) (relativeAddress >> 8));
        }
      }
    }
```

The **GenerateReturn** method changes the return address from the address relative the beginning of the function to the address relative the beginning of the executable code. Similar to the **GenerateAccess** case above, we need to inspect the original address, and add the address of the beginning function.

```
    private void GenerateReturn(int functionStartAddress, ISet<int>
returnSet,
                                List<byte> byteList) {
      foreach (int sourceAddress in returnSet) {
        int lowByte = byteList[sourceAddress],
            highByte = byteList[sourceAddress + 1];
        int targetAddress = (highByte << 8) + lowByte;
        targetAddress += functionStartAddress;
        byteList[sourceAddress] = (byte) targetAddress;
        byteList[sourceAddress + 1] = (byte) (targetAddress >> 8);
      }
    }
```

The **SimpleName** method returns the part to the left of the leftmost separator identity characters (‘$’), if present. Otherwise, the whole string is returned.

```
    public static string SimpleName(string name) {
      int index = name.LastIndexOf(Symbol.SeparatorId);
      return (index != -1) ? name.Substring(0, index) : name;
    }
  }

}
```