

You are to implement a two-pass linker in C, C++, or Java and submit the **source** code on eClasses, which we compile and run on the EECS red server.

*This assignment is adopted from the lab designed by Prof Allan Gottlieb*

## Linker

As we discussed in the last lecture, for executing a computer program code it needs to be compiled (by a Compiler) into assembly. The assembler processes the code to produce machine instructions into the **object module**. The object module cannot be run because the compiler and assembler produce each module as if it will be loaded at location zero. These modules need to be *linked* together to produce consistent memory map. This is what the linker does. Linker is a utility program that implements two tasks: 1) **relocating relative addresses** and 2) **resolving external references**.

### Relocating Relative Addresses

Remember each module is produced starting from zero. For example, a machine instruction representing:

```
JMP 120
```

is used to indicate a jump to location 120 of the current module. To convert this **relative address** to an **absolute address**, the linker adds the **base address** of the module to the relative address. The base address is the address at which this module will be loaded.

For example, assume a module is to be loaded starting at location 2300 and contains the above instruction

```
JMP 120
```

The linker changes this instruction to

```
JMP 2420
```

How does the linker know that the module is to be loaded starting at location 2300?

- Linker processes the modules one at a time: We assume that the first module is to be loaded at location zero. So relocating the first module is trivial (adding zero). We say that the relocation constant is zero.
- After processing the first module, the linker knows its length (say that length is L1).
- Hence the second module is to be loaded starting at L1, i.e., the relocation constant is L1.
- In general, the linker keeps the sum of the lengths of all the modules it has already processed; this sum is the relocation constant for the next module.

### Resolving External References

If a module contains a function call  $f(x)$  to a function  $f()$  that is defined in a **different** module, the object module containing the call must contain some kind of jump to the beginning of  $f()$

- But this is impossible! When the program is compiled, the compiler and assembler do not see the definition of  $f()$  so there is no way they can supply the starting address.
- Instead a dummy address is supplied and a notation made that this address needs to be filled in with the location of  $f()$ . This is called a **use** of  $f()$ .
- The object module containing the **definition** of  $f()$  contains a notation that  $f()$  is being defined and gives the relative address of the definition, which the linker converts to an absolute address (as above).
- The linker then changes all uses of  $f()$  to the correct absolute address.

## Assignment 2: Two Pass Linker

For your assignment you will simulate a two pass linker. The target machine is word addressable and has a memory of 300 words, each consisting of 4 decimal digits. The first (leftmost) digit is the opcode, which is unchanged by the linker. The remaining three digits (called the address field) form either

- An immediate operand, which is unchanged.
- An absolute address, which is unchanged.
- A relative address, which is relocated.
- An external address, which is resolved.

### I/O specification

There are several sample input sets attached below. The first is shown below and the second is an re-formatted version of the first. If you use the Java Scanner or C's scanf() (which I recommend you do), inputs 1 and 2 will look the same to your program. Some of the input sets contain errors that you are to detect as described below. We will run your assignments on these (and other) input sets.

The input consists of a series of object modules, each of which contains three parts: *definition list*, *use list*, and *program text*. Preceding all the object modules is a *single integer* giving the number of modules present.

The linker processes the input twice (that is why it is called two-pass). Pass one determines the base address for each module and the absolute address for each external symbol, storing the later in the symbol table it produces. The first module has base address zero; the base address for module  $i + 1$  is equal to the base address of module  $i$  plus the length of module  $i$ . The absolute address for a symbol  $S$  defined in module  $M$  is the base address of  $M$  plus the relative address of  $S$  within  $M$ . Pass two uses the base addresses and the symbol table computed in pass one to generate the actual output by relocating relative addresses and resolving external references.

The definition list is a count  $ND$  (Number of Definitions) followed by  $ND$  pairs  $(S, R)$  where  $S$  is the symbol being defined and  $R$  is the relative address to which the symbol refers. Pass one relocates  $R$  forming the absolute address  $A$  and stores the pair  $(S, A)$  in the symbol table.

The use list is a count  $NU$  (Number of Uses) followed by  $NU$  pairs  $(S, R)$ , where  $S$  is an external symbol used in the module and  $R$  is a relative address where  $S$  is used. The (dummy) address initially in  $R$  is a pointer to the next use of  $S$ . This linked list of uses ends with a pointer of 777.

The program text consists of a count  $NT$  (Number of Text entries) followed by  $NT$  pairs  $(type, word)$ , where  $word$  is a 4-digit instruction as described above and  $type$  is a single character indicating if the address in  $word$  is Immediate, Absolute, Relative, or External.  $NT$  is also the length of the module.

The actions taken by the linker depend on the type of the address. Consider the following input:

A program text consists of a count NT (Number of Text entries) followed by NT pairs (type, word), where word is a 4-digit instruction as described above and type is a single character indicating if the address in word is Immediate, Absolute, Relative, or External. NT is also the length of the module.

The actions taken by the linker depend on the type of the address. Consider the following input:

```
4
1 xy 2
2 xy 4 z 2
5 R 1004 I 5678 E 2777 R 8002 E 7777
0
1 z 3
6 R 8001 E 1777 E 1001 E 3002 R 1002 A 1010
0
1 z 1
2 R 5001 E 4777
1 z 2
1 xy 2
3 A 8000 E 1777 E 2001
```

In the first pass, the linker simply finds the base address of each module and produces the symbol table giving the values for xy and z (2 and 15 respectively). The second pass does the real work using the symbol table and base addresses produced in pass one.

The resulting output (shown below) is more detailed than I expect you to produce (for an example, see the attached outputs). The detail is there to help me explain what the linker is doing.

The following is output annotated for clarity and class discussion. Your output is not expected to be this fancy.

```
Symbol Table
xy=2
z=15

Memory Map
+0
0: R 1004 1004+0 = 1004
1: I 5678 5678
2: xy: E 2777 ->z 2015
3: R 8002 8002+0 = 8002
4: E 7777 ->xy 7002
+5
0 R 8001 8001+5 = 8006
1 E 1777 ->z 1015
2 E 1001 ->z 1015
3 E 3002 ->z 3015
4 R 1002 1002+5 = 1007
5 A 1010 1010
+11
0 R 5001 5001+11= 5012
1 E 4777 ->z 4015
+13
0 A 8000 8000
1 E 1777 ->xy 1002
2 z: E 2001 ->xy 2002
```

#### Other requirements: Error detection, arbitrary limits, et al.

Your program must check the input for the errors listed below. All error messages produced must be informative, e.g., "Error: X21 was used but not defined. It has been given the value 111".

- If a symbol is multiply defined, print an error message and use the value given in the last definition.
- If a symbol is used but not defined, print an error message and use the value 111.
- If a symbol is defined but not used, print a warning message.
- If an absolute address exceeds the size of the machine, print an error message and use the largest legal value.
- If multiple symbols are listed as used in the same instruction, print an error message and ignore all but the last usage given.
- If a type R address in the list of Text entries exceeds the size of the module, treat the address as 0 (and relocate it since it is of type R).

You may need to set "arbitrary limits", for example you may wish to limit the number of characters in a symbol to (say) 8. Any such limits should be clearly documented in the program and if the input fails to meet your limits, your program must print an error message. Naturally, any such limits must be large enough for all the attached inputs. *Under no circumstances should your program reference an array out of bounds, etc.*

## What to submit

You need to submit the following a zip file (named: `firstname_lastname_yorkid.zip`) that includes the following required components :

- individual C/C++ or Java implementation (source files) of the linker, named linker (and other dependency source files the required to run your compile and run your program)
  - Your program should be executed using for C/C++: `./linker <input_file.txt>` or Java: `java linker <input_file.txt>`
    - Inputs from standard input, i.e., requiring TAs to type in the input, will not be accepted.
- Your program must read its input from a file, whose name is given as a command line argument and send its output to the screen (`printf()` in C; `System.out` in Java). NOTE:
  - The inputs will be correct per specification but can be in different formats (see input 1 and 2 in the attached IO pairs)
  - Make sure your outputs match the format of the provided output files.
- Each file should include the following header:

```
/**
 * Full Name:
 * Course ID:
 * Description: a very brief description of what is implemented (no more than 4 lines)
 */
```
- You must include enough high-level comments in all files of your program so that a reader (e.g., the grader) who is expert in C/C++/Java and knowledgeable about the two-pass linker algorithms can understand the basic operation of your program.
- Your code should be properly formatted (use can use online services (e.g., codebeautify) this one to achieve this)
- You must include a README.txt file describing each included files, provides an overall brief description of what program does and provides instructions on compiling and running the assignment on the EECS red server.
- If you're using C or C++ to implement the assignment, you must include a **Makefile** which will be used to compile and run your program
- Note: TAs are instructed to deduct points for programs that don't follow the above guidelines and require special adjustment to test (e.g., due to non-trivial compilation and execution, non-conforming to IO formats, etc).