In this lab you will be implementing a two-pass linker. In general, a linker takes individually compiled code/object modules and creates a single executable by resolving external symbol references (e.g. variables and functions) and module relative addressing and assigning global addresses after placing the modules' object code at global addresses.

Rather than dealing with complex x86 tool chains that I demonstrated in class, we assume a target machine with the following properties: (a) word addressable, (b) addressable memory of 512 words, and (c) each valid word is represented by an integer (<10000). *I know that is a really strange machine*, but core tasks of linkers such relocation, relative addressing, symbol table maintenance and in particular error checking and reporting is handled in this notation as well.

The input to the linker is a file containing a **sequence of tokens** (symbols and integers and instruction mode characters). Don't assume tokens that make up a section to be on one line, nor make assumptions about how much space separates tokens or that lines are non-empty for that matter or that each input conforms syntactically. Symbols always begin with alpha characters followed by optional alphanumerical characters, i.e.[a-Z][a-Z0-9]*. Valid symbols can be up to 16 characters. Integers are decimal based. Instruction type characters are (M, A, R, I, E). Token delimiters are ' ', '\t' or '\n'.

The input file to the linker is structured as a series of "object module" definitions.
Each "object module" definition contains three parts (in fixed order): definition list, use list, and program text.

- ***definition list*** consists of a count *defcount* followed by *defcount* pairs (S, R) where S is the symbol being defined and R is the relative word address (offset) to which the symbol refers in the module (0-based counting).
- ***use list*** consists of a count *usecount* followed by *usecount* symbols that can be referred to in this module. These could include symbols defined in the *definition list* of any module (prior or subsequent or not at all).
- ***program text*** consists of a count *codecount* followed by *codecount* pairs (**addrmode, instr**), where *addrmode* is a single character indicating the addressing mode as **M**odule, **A**bsolute**, R**elative, **I**mmediate or **E**xternal. *instr* is the instruction (integer). Note that *codecount* defines the length/size of the module.

The instruction integer is comprised of an opcode (instruction/1000) and an operand (instruction%1000). The opcode always remains unchanged by the linker. For the instruction value read an integer and ensure opcode < 10, see errorcodes below. The operand is modified/retained based on the instruction type in the *program text* as follows:
(M) operand is the number of a valid module and is replaced with the base address of that module.
(**A**) operand is an absolute address which will never be changed in pass2; however it can't be ">=" the machine size (512);
(**R**) operand is a relative address in the module which is relocated by replacing the relative address with the absolute address of that relative address after the module's global address has been determined (absolute_addr = module_base+relative_addr).
(**I**) an immediate operand is unchanged, but must be less than 900.
(**E**) operand is an external address which is represented as an index into the uselist. For example, a reference in the program text with operand K represents the Kth symbol in the use list, using 0-based counting, e.g., if the use list is ''2 f g'', then an instruction ''E 7000'' refers to f, and an instruction ''E 5001'' refers to g. You must identify to which global address the symbol is assigned and then replace the operand with that global address.

The linker must process the input twice (that is why it is called two-pass) (to preempt the favored question: "Can I do it in one pass?" → NO, because storing tokens makes your program more complex). **Pass One** parses the input and verifies the correct syntax and determines (1) the base address for each module and stores it in a module_base table, and (2) the absolute address for each defined symbol, storing the latter in a symbol table. The first module has base address zero; the base address for module X+1 is equal to the base address of module X plus the length of module X (defined as the number of instructions in a module). The absolute address for symbol S defined in module M is the base address of M plus the relative address of S within M. After pass one, print the symbol table (including errors related to it (see rule2 later) *in order of appearance*). Do not store parsed tokens, the only data you should and need to store between passes are the module_base table the symbol table.
Pass Two again parses the input and uses the base addresses and the symbol table entries created in pass one to generate the actual output by relocating relative addresses and resolving external references. You should reuse pass-1 parser code just with different actions. You must clearly mark your two passes in the code through comments and/or proper function naming.

While parsing you cannot store tokens, not between pass one and pass two, nor while parsing a single line. This practices reentrant code which one encounters frequently in Operating Systems. This implies you can only parse and consume one token at a time and can't run through a full line and store tokens per line or file as indicated in the strtok() man page. To enforce this we deduct 3 pts if you do this.

Other requirements: error detection, limits, and space used.

To receive full credit, you must check the input for various errors (test inputs will have lots of errors). All errors/warnings should follow the message catalog provided below. We will do a textual difference against a reference implementation to grade your program. Any reported difference will indicate a non-compliance with the instructions provided and is reported as an error and results in deductions.
You should continue processing after encountering an error/warning (other than a syntax error) and you should be able to detect multiple errors in the same run.

1. You should stop processing if a syntax error is detected in the input, print a syntax error message with the line number and the character offset in the input file where observed. A syntax error is defined as a missing token (e.g. 4 used symbols are defined but only 3 are given) or an unexpected token. Stop processing and exit.
2. If a symbol is defined multiple times, print an error message and use the first definition. The error message is to appear as part of printing the symbol table (following symbol=value printout on the same line).
3. If a symbol is used in an E-instruction but not defined anywhere, print an error message and use the value absolute zero.
4. If a symbol is defined but not used in an E instruction, print a warning message and continue.
5. If an address appearing in a first definition of a symbol exceeds the size of the module, print a warning message in pass1 after processing the module and treat the address given as 0 (relative to the module). If the symbol is a redefinition then print a warning message (see message below).
6. If an external operand is too large to reference an entry in the use list, print an error message and treat the operand as relative=0.
7. If a symbol appears in a use list but is not actually used in the module (i.e., not referred to in any E-type address), print a warning message and continue. If the same unused symbol appears multiple times in uselist, multiple warnings are to be printed.
8. If an absolute address exceeds the size of the machine, print an error message and use the absolute value zero.
9. If a relative address exceeds the size of the module, print an error message and use the module relative value zero (that means you still need to remap "0" that to the correct absolute address).
10. If an illegal immediate operand (I) is encountered (i.e. >= 900), print an error and convert the operand value to 999.
11. If an illegal opcode is encountered (i.e. op >= 10), print an error and convert the <opcode,operand> to 9999.
12. If a module operand is invalid, print an error message and assume module 0.

The following exact limits are in place.

a) Accepted symbols should be upto 16 characters long (not including terminations e.g. '\0'), any longer symbol names are erroneous.
b) a uselist or deflist should support 16 definitions, but not more and an error should be raised.
c) number of instructions are unlimited (hence the two pass system), but in reality they are limited to the machine size.
d) Symbol table should support at least 256 symbols (reference program supports exactly 256 symbols).
e) Module table should support at least 128 entries (reference program supports exactly 128 entries).

There are several sample inputs and outputs provided as part of the sample input files / output files (see NYU Classes).
The first (*input-1)* is shown below and the second (*input-2)* is a re-formatted version of the first. They both produce the same output as **the input is token-based** and hence present the same content to the linker. Many of the input sets contain errors that you are to detect as described above. Note that when you have questions regarding errors, please first make sure the structure of the input is not messing with your mind. We will run your lab on these (and other) input sets.

```
1 xy 2
2 z xy
5 R 1004  I 5678  E 2000  R 8002  E 7001
0
1 z
6 R 8001  E 1000  E 1000  E 3000  R 1002  A 1010
0
1 z
2 R 5001  E 4000
1 z 2
2 xy z
3 A 8000  E 1001  E 2000
```

Your output is expected to strictly follow this format (with exception of empty lines):

```
Symbol Table
xy=2
z=15

Memory Map
000: 1004
001: 5678
002: 2015
003: 8002
004: 7002
005: 8006
006: 1015
007: 1015
008: 3015
009: 1007
010: 1010
011: 5012
012: 4015
013: 8000
014: 1015
015: 2002
```

The following output (`./linker -e input-1`) is heavily annotated for clarity and class discussion. You are not creating this output. However, it should help you understand the operation and mapping of symbols etc.

```
Symbol Table
xy=2
z=15

Memory Map| Label | Instr  -> Explanation  | Warn/Errors
----------+-------------------------------------------
@Module_0
000: 1004 |       : R 1004 -> base=0+4     |
001: 5678 |       : I 5678                 |
002: 2015 | xy    : E 2000 -> z=15         |
003: 8002 |       : R 8002 -> base=0+2     |
004: 7002 |       : E 7001 -> xy=2         |
@Module_1
005: 8006 |       : R 8001 -> base=5+1     |
006: 1015 |       : E 1000 -> z=15         |
007: 1015 |       : E 1000 -> z=15         |
008: 3015 |       : E 3000 -> z=15         |
009: 1007 |       : R 1002 -> base=5+2     |
010: 1010 |       : A 1010                 |
@Module_2
011: 5012 |       : R 5001 -> base=11+1    |
012: 4015 |       : E 4000 -> z=15         |
@Module_3
013: 8000 |       : A 8000                 |
014: 1015 |       : E 1001 -> z=15         |
015: 2002 | z     : E 2000 -> xy=2         |
```

Note, that even an empty program should have the “Symbol Table” and “Memory Map” line.

For a test case to pass you must catch ALL warning/errors and generate the correct output for a given input file.

Example:

```
Symbol Table
X21=3
X31=4

Memory Map
000: 1003
001: 1003
002: 1003
003: 2000  Error: Absolute address exceeds machine size; zero used
004: 3000  Error: Relative address exceeds module size; relative zero used

Warning: Module 3: X31 was defined but never used
```

Parse errors should abort processing at the time of occurrence.
Error messages must be following the instruction as shown above.
Warnings message locations are defined further down.
Module counting starts at 0.

I provide in C the code to print parse errors, which also gives you an indication what is considered a parse error.

```
void __parseerror(int errcode) {
    static char* errstr[] = {
        "NUM_EXPECTED",              // Number expect, anything >= 2^30 is not a number either
        "SYM_EXPECTED",              // Symbol Expected
        "MARIE_EXPECTED",            // Addressing Expected which is M/A/R/I/E
        "TOO_MANY_DEF_IN_MODULE",    // > 16
        "TOO_MANY_USE_IN_MODULE",     // > 16
        "TOO_MANY_INSTR",             // total num_instr exceeds memory size (512)
        "SYM_TOO_LONG",              // Symbol Name is too long
    };
    printf("Parse Error line %d offset %d: %s\n", linenum, lineoffset, errstr[errcode]);
    exit(1);
}
```

(Note: line numbers start with one and offsets in the line start with one, offsets should indicate the first character offset of the token that is wrong, not the last). Tabs (‘\t’) count as one character.
Error messages have the following text and should appear right at the end of the line you are printing out

"Error: This variable is multiple times defined; first value used"	(see rule 2)
"Error: %s is not defined; zero used" (insert the symbol name for %s)	(see rule 3)
"Error: External operand exceeds length of uselist; treated as relative=0"	(see rule 6)
"Error: Absolute address exceeds machine size; zero used"	(see rule 8)
"Error: Relative address exceeds module size; relative zero used"	(see rule 9)
"Error: Illegal immediate operand; treated as 999"	(see rule 10)
"Error: Illegal opcode; treated as 9999"	(see rule 11) takes precedent
"Error: Illegal module operand ; treated as module=0"	(see rule 12)

Warning messages have the following text and are on a separate line.

"Warning: Module %d: %s=%d valid=[0..%d] assume zero relative”	(see rule 5) [see input-10]
“Warning: Module %d: %s redefinition ignored\n"	(see rule 5) [see input-4]
"Warning: Module %d: uselist[%d]=%s was not used\n"	(see rule 7)
"Warning: Module %d: %s was defined but never used\n”	(see rule 4)

Locations for these warnings are:

Rule 5: to be printed after each module in pass1
Rule 7: to be printed after each module in pass2 (so actually interspersed in the memory map (see out-9)
Rule 4: to be printed after pass 2 (i.e. all modules have been processed) (see out-3).

Parse Error Location:

Parse errors are to be located at the first character of the wrong token, or if end-of-file is reached at the end of file location. There is one special case when the eof ends with `\n'. My expectation is that the line number reported actually exists in the file and that an editor (e.g. vi) can jump to it. In this particular case the linenumber to be reported is the last line read and the last position of that line, not the next line and offset 1 (see *input-12* for an example). The error is at the very last position of the line. Reason is when one does a linecount on the file ("wc –l input-12") it shows 3 not 4.

Hint: for each parser error and warning error mentioned above you should have code checking for that.

Program Specification

Only C/C++ are allowed for this lab or any subsequent labs. The program should take a single command line argument for input file. The output of the program must go to **standard output.**
The program **should be run** on any of the **crunchy[1,2,5,6].cims.nyu.edu** machines of NYU-CIMS, which is where it will be graded, so please make sure your program runs there.

Testing your program before submission

As part of the *lab1_assign.tar.Z* in NYU Brightspace you will see a ***runit.sh*** and a ***gradeit.sh*** script (the same we will use for grading albeit with more inputs). Use "tar -xzvf lab1_assign.tar.Z" to decompress the file on a Unix machine. Understand the script and what it does. If you can't pass this script, things will be flagged during the grading process as well. Don't assume that just because you pass for these inputs, it will pass for all inputs. Carefully go through the rules above and ensure covered each rule with some code (if/then/else) and it is at the right location.

Execute as follows: Create yourself a directory "outdir" where your outputs will be created. Replace cmds with the name.

```
> cd lab1_assign
> ./runit.sh "outdir" <your-executable and optional arguments>
```

The above will create all the outputs for the available inputs (1-20) in "outdir" which has to be created manually.

```
> ./gradeit.sh . "outdir" # note 2nd argument is a DOT for the local directory where reference output are.
```

The above will compare the reference outputs (out-[1-20]) with the ones created with your program and tell you how many you got right and which ones are wrong. There will be a file called <your-outdir>/LOG.txt that contains which cases you got wrong and where the differences are. If you want to analyze further, manually run "`diff -b -B -E`" by hand on a particular output pair and rinse and repeat till you have got a test case right.

The reference program used during grading is located on */home/hf44/Public/lab1/linker* on cims systems. So feel free to try it in order to answer any questions you might have on what is expected for a particular input. An input generator is provided under /home/hf44/Public/lab1/lab1gen.py

What to submit (this applies to ALL labs)

Submit a zip archive containing only (i) source code (ii) makefile to compile your code (iii) the make output log and (iv) the gradeit.sh output log. (iii) and (iv) must come from a crunchy cims machine. Please make sure the zip does not have any input and/or output files nor contains other zip or tar files or backup files. We accept *.zip *.tar and *.tar.Z files (as long as they match how they were created. Anything else will be returned for resubmission (we use download tools based on those formats) with 1 pts deduction.

(iii) execute : `( hostname; make clean; make ) > make.log`
as you execute in your src dir, you have it in the right place

(iv) execute : `./gradeit.sh . outdir > gradeit.log`
note "." Is the local directory and outdir is the directory you created and used in runit.sh

Grading

This lab is graded using a "diff –b –B" against the reference output created by the test program (aka reference program) using a grading harness described above (*runit.sh + gradeit.sh)*.
Inputs will be the ones provided in NYU Classes as well as other ones and will be checked for all of the error conditions.
It is imperative that you match the output as generated by the ref program to allow for the automated testing for yourself as well. Use the harness instead of manual checking when you have the basics running, it tells you where things are wrong.
We score this lab as 100pts. You will receive 40 pts for a submission that attempts to solve the problem. The rest you get 60/N points for each successful test that passes the "*diff*". In order to institute a certain software engineering discipline, i.e. following a specification and avoiding unintended releases of code and data in real life, we account for the following additional deductions:

Reason	**Deduction**	**How to avoid**
Late submission	2pts/day	Upto 7 days late allowed. No more submissions afterwards.
Makefile not working on CIMS or missing.	1pt	Make sure your source compiles with your Makefile by simply typing "make" and "make clean"
CIMS Make.log missing	1pts	Run it on CIMS and include in zip
CIMS gradeit.log missing	4pts	Run it on CIMS and include in zip
Minor code fixes	<=4pts	Verify build and run on CIMS
Not parsing tokens one at a time and storing Tokens instead of reparsing the file or storing Tokens when parsing a line. All you should store is the symbol and module tables between passes.	3pts	Follow instructions on parsing. After that, copy the shell of the parser for the 2nd pass. Close the file, reopen the file and parse again, just change the actions taken between 1st (error checking, module & symbol table creation) and 2nd pass (instruction transformation).
Inputs/Outputs or *.o files in the submission	1pt	Go through your intended submission and clean it up.
Output not going to the screen but to a file	1pt	We utilize the output to <stdout> during the runit.sh and gradeit.sh so just use printf or cout. You will have to fix this
Incorrect Format submitted	1pt	Use zip/tar tools to create a submission

How to approach this lab:

Step 1: Write a tokenizer
Write a tokenizer that simply parses the input, prints the token and the position in the file found, at the end prints the final position in the file (as you will need that for error reporting). Verify it correctly recognizes tokens, lines and line offsets and also print the final error location as the last line read with last character in the line (not the next empty line).
Historically, getting this first step right and then layering <step 2> over it is the main headache in this assignment.
The functions you need to study for this are `getline()` or `fgets()` (reads a full line into a buffer) or C++ equivalent and `strtok()`, which tokenizes input lines. `Please` use Linux's build-in help : "man strtok" and understand how a new line is seeded and continued in subsequent calls. My tokenizer executable is available on cims under /home/hf44/Public/lab1/tokenizer (see what it produces on an input). You don't have to match, its just a good way to start. Read the "man page"; it will tell you all you need to know how to use strtok (including examples and what to watch out for).

Your optional **tokenizer** program should probably look similar to the following and you will use the getToken() in step2+. Don't use the code you find in the `man strtok`, it will lead you down the wrong path if taken as is.

```
main() {
       char *tok;
       while( (tok = getToken()) != NULL ) {
                 printf("token=<%s>  position=%d:%d\n", tok, linenum, lineoffset);
       }
       printf("EOF position %d:%d\n", linenum, lineoff);
}
```

Step 2: Write the basic token functions
Once you have the `getToken()` function written above and you verified that your token locations are properly reported, extract it out of the program above and start writing the linker program. Layer the `readInt(), readSymbol(), readMARIE()` functions on top of it by checking that the token has the correct sequence of characters and length (e.g. integers are all numbers) and test via a simple program. Macros like `isdigit(), isalpha(), isalnum()` can proof useful.

Step 3: Write the parser
Here are some hints on writing a parser. In general, one could write this parser using lex/yacc, however this is so simple that I suggest writing a simple recursive decent parser, in particular since you have to parse the input twice. In lex and yacc you would need to handle the error locations to conform to the specification. You would have a structure similar to the following (all pseudoCode). This structure is simply copied twice and the actions taken in each path are different.
Once you have the first pass written, reset the input file, copy the pass1() to pass2() and rewrite. Note, in pass2 certain error checking doesn't have be done anymore. Instead, you are now rewriting the instructions.

```
Pass1() {
      while (true) {
            // this is a new model
            int defcount = readInt(); // return negative to indicate no more tokens
            if (defcount < 0) exit(2); // eof reached
            for (int i=0;i<defcount;i++) {
                 Symbol sym = readSym();
                 int val = readInt();
                 createSymbol(sym,val);              ← this would change in pass2
            }
            int usecount = readInt();
            for (int i=0;i<usecount;i++) {
                 Symbol sym = readSym();
                 // we don't do anything here        ← this would change in pass2
            }
            int instcount = readInt();
            for (int i=0;i<instcount;i++) {
                 char addressmode = ReadMARIE();
```

```
                int  operand = ReadInt();
                :  // various checks                ← this would change in pass2
                :  //  - " -
            }
        }
}
```

Of course, you will have to deal with the errors etc. The first pass does syntax and early error checking and creates the symbol and module table, the second pass performs additional error checking and instruction transformation. In order to go over the file a second time reset the file pointer or close and reopen the file. **You should not store tokens or deflist or uselist or instructions** during the first pass in order to pass them to pass2. Only the symbol table and module table needs to be stored between passes. Note while parsing a single module you do temporarily have to store the uselist and deflist.

Writing a Makefile

A makefile should be named either "makefile" or "Makefile". *make* will look for either of these files during building. Here is a good introduction: *http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/* with more sophisticated ones. The simplest Makefile you can think of contains how it builds the executable and how it cleans up (here it assumes one source file only and builds the *linker* executable) and is perfectly OK. Make your life and ours simple:

```
linker: linker.cpp
        g++ -g linker.cpp -o linker   # I always compile with -g to enable debugging

clean:
        rm -f linker *~
```

Here is a more sophisticated one that assumes multiple inputfiles: file1.c .. file3.c and allows to specify the compiler version

```
CFLAGS=-g        # -g = debug,  -O2 for optimized code    CXXFLAGS for g++

linker: file1.c file2.c file3.c
        $(CC)  $(CFLAGS) -o linker file1.c file2.c file3.c

clean:
        rm -f linker *~
```

Hint: The "makefile" is typically the first file I create and edit in any project requiring compilation.

Debugging help:

You can run the reference program with the -e or -E option to get explanations. Don't ignore warnings potentially add "-Wall" to your CFLAGS or CXXFLAGS and fix those warnings.

It is imperative that you test your code on crunchy[1,2,5,6] servers and run (runit.sh and gradeit.sh). If there are errors, go to the LOG.txt file created in the output directory and see what's wrong and fix it. Note that code that runs fine on windows or MAC might not run correctly on Linux machines due to differences in compiler and memory layout and semantics of variable initialization etc.

The burden is on you, we will grade on a crunchy machine only.

GOOD LUCK