**RIZVI COLLEGE OF ENGINEERING**

DEPARTMENT OF COMPUTER ENGINEERING

LAB MANUAL

SESSION 2018-19

SUBJECT: SYSTEM SOFTWARE LAB

YEAR/SEMESTER: 3RD/6TH

**Objectives:**

1. To help students appreciate the role and functioning of various system programs over application program

2. To initiate an understanding of compilers in general and brief about phases of compiler.

3. To provide a theoretical framework for optimizing the code.

4. To familiarize and encourage the students to use various software tools for Developing System programs.

Outcomes: Learner will be able to…

1. Identify different system software

2. Use Lex tool used for generating lexical analyzer.

3. Write macros as and when required to increase readability and productivity

4. Design hand written lexical analyzer

5. Design new language structures with the help of grammars

6. Appreciate the role of Operating System functions such as memory management as pertaining to run time storage management

7. Appreciate role of Intermediate Code Generation in connection with language designing

8. Apply optimization principles on given code

9. Implement various parser types and use YACC.

**LIST OF EXPERIMENTS:**

1. Implementation Of Lexical Analyser Using Lex tool Lexical analyzer tool : flex

2. Syntax Analysis Using Yacc Parser generator tool : Yacc.

3. Find first() , follow() set of given grammer

4. removing left recursion direct as well as indirect given the set of production rule

5. Assemblers : 2 pass Assembler .

6. Macroprocessor : 2 pass Macro processor.

7. Syntax Analysis : ( any 1 of LL(1) , LR(0) , LR(1) , LALR(1) , operator precedence parser. )

8. Create your library in Linux environment and using it.

9. Code Generation algorithm.

10.Code Optimization techniques.

11.Study ld command in Unix/Linux.

EX.NO:1    IMPLEMENTATION OF LEXICAL ANALYSER USING LEXTOOL  
  
AIM:   
    To write a C program to implement lexical analysis using LEX tool.  
  
ALGORITHM:  
  
1.    Start the program.  
2.    Lex program consists of three parts.  
a.    Declaration         %%  
b.    Translation rules %%  
c.    Auxilary procedure.  
3.    The declaration section includes declaration of variables, maintest, constants and regular definitions.  
4.    Translation rule of lex program are statements of the form  
a.    P1 {action}  
b.    P2 {action}  
c.    …  
d.    …  
e.    Pn {action}  
5.    Write a program in the vi editor and save it with .l extension.  
6.    Compile the lex program with lex compiler to produce output file as lex.yy.c. eg  $ lex filename.l   
     $ cc lex.yy.c -ll  
7.    Compile that file with C compiler and verify the output.  
  
PROGRAM:  
%{  
%}  
identifier[a-zA-Z][a-zA-Z0-9]\*  
%%  
#.\*      printf("\n%s is PREPROCESSOR DIRECTIVE\n", yytext);  
int  |  
float |  
double |  
char |  
for |  
  
if printf("%s is a keyword\n",yytext);  
{identifier}\( printf("\n\n FUNCTION CALL\n %s",yytext);  
\{  
printf("BLOCK BEGINS\n");  
\}  
printf("BLOCK ENDS\n");  
{identifier}(\[[0-9]\*\])? printf("%s is identifier\n",yytext);  
= printf("%s is a ASSIGNMENT OPERATOR\n",yytext);  
[0-9]+  printf("%s is NUMBER\n",yytext);  
\< |  
\> |  
\== |  
\>= |  
\<=  printf("%s is a RELATIONAL OPERATOR\n",yytext);  
\( { ECHO(;printf("\n");}  
\) { ECHO); printf("\n");}  
\+ |  
\- |  
\\*  printf("%s is a ARITHMETIC OPERATOR \n");  
\++  printf("%s is a INCREMENTAL OPERATOR\n");  
\;  { ECHO; printf("\n");}  
%%  
main()  
{  
yylex();  
}  
int yywrap()  
{  
return 1;  
}  
"sam1.l" 42L, 753C written  
  
  
  
  
  
  
  
  
  
OUTPUT:  
  
[localhost@server ~]$ lex sam1.l                        
[localhost @server ~]$ cc lex.yy.c -ll                 
[localhost @server ~]$. /a. out   
a  
a is identifier  
  
#  
  
# is PREPROCESSOR DIRECTIVE  
  
include  
include is identifier  
  
<  
< is a RELATIONAL OPERATOR  
  
stdio  
stdio is identifier  
  
  
  
  
  
  
RESULT:  
           
Thus the lexical analyser is implemented using lex tool.

EX. NO: 2   SYNTAX ANALYSIS USING YACC  
  
AIM:  
  
       To implement simple desk calculator using Yacc tool.  
  
ALGORITHM:  
1.    Start the program.  
2.    Yacc program consists of three parts namely  
a.    Declarations   
%%  
b.    Transition Rule  
%%  
c.    Supporting C – routines.  
3.    Declaration part consists of two sections, first section contains only include statements and the second statements contains declaration of the grammar tokens.  
4.    Each rule in set of transition rules consists of grammar production and semantic action. The set of productions are of the form  
a.    <left side>: <alt 1> {semantic action 1}    
1.    | <alt 2> {semantic action 2}  
2.    …..  
3.    | <alt n> {semantic action n}  
4.    ;  
5.    In the third part, error recovery routines are added.  
6.    The program is typed using vi editor, and saved with .y extension.  
7.    It is first compiled with the yacc compiler to produce the C code for C compiler (yacc samp.y).  
8.    After that compile that program with C compiler (cc y.tab.c – ly standard output file of yacc compiler).  
9.    See the output using ./a. out.  
10.    Stop the program.  
  
PROGRAM:  
%{  
#include<ctype.h>  
#include<stdio.h>  
#define YYSTYPE double  
%}  
  
  
%token NUMBER  
%left '+' '-'  
%left '\*' '/'  
%right '^'  
%right UMINUS  
%%  
  
lines : lines expr '\n' {printf("%lf",$2);}  
        | lines '\n'  
        |  
        ;  
expr : expr '+' expr {$$=$1+$3;}  
| expr '-' expr {$$=$1-$3;}  
| expr '\*' expr {$$=$1\*$3;}  
| expr '/' expr {if($3!=0) {$$=$1/$3;}  
else yyerror("divident should be a positive no.\n");  
}  
| expr '^' expr { int i,sum=1;  
                for(i=1;i<=$3;i++)  
                        {sum \*= $1;}  
                $$=sum;  
                }  
| '(' expr ')'  {$$=$2;}  
| '-' expr %prec UMINUS {$$= $2;}  
| NUMBER  
;  
%%  
yylex()  
{  
int c;  
while((c=getchar())==' ');  
if((c=='.')||(isdigit(c)))  
{  
ungetc(c,stdin);  
scanf("%lf",&yylva l);  
return NUMBER;  
}  
return c;  
}  
  
OUTPUT:  
  
[localhost@server ~]$ yacc samp.y  
[localhost@server ~]$ cc y.tab.c -ly  
[localhost@server ~./a.out  
9  
9.000000  
6+3  
9.000000  
25-12  
13.000000  
  
RESULT: Thus a calculator is implemented using YACC tool.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**EX.NO:3**   **Find first() , follow() set of given grammer**

**AIM**: To write a C program to implement Find first() , follow() set of given grammer

**Theory**: The functions follow and followfirst are both involved in the calculation of the Follow Set of a given Non-Terminal. The follow set of the start symbol will always contain “$”. Now the calculation of Follow falls under three broad cases :

* If a Non-Terminal on the R.H.S. of any production is followed immediately by a Terminal then it can immediately be included in the Follow set of that Non-Terminal.
* If a Non-Terminal on the R.H.S. of any production is followed immediately by a Non-Terminal, then the First Set of that new Non-Terminal gets included on the follow set of our original Non-Terminal. In case encountered an epsilon i.e. ” # ” then, move on to the next symbol in the production.  
  **Note :** “#” is never included in the Follow set of any Non-Terminal.
* If reached the end of a production while calculating follow, then the Follow set of that non-teminal will include the Follow set of the Non-Terminal on the L.H.S. of that production. This can easily be implemented by recursion.

**Assumptions :**

1. Epsilon is represented by ‘#’.
2. Productions are of the form A=B, where ‘A’ is a single Non-Terminal and ‘B’ can be any combination of Terminals and Non- Terminals.
3. L.H.S. of the first production rule is the start symbol.
4. Grammer is not left recursive.
5. Each production of a non terminal is entered on a different line.
6. Only Upper Case letters are Non-Terminals and everything else is a terminal.
7. Do not use ‘!’ or ‘$’ as they are reserved for special purposes.

**Explanation :**  
Store the grammar on a 2D character array production. findfirst function is for calculating the first of any non terminal. Calculation of first falls under two broad cases :

* If the first symbol in the R.H.S of the production is a Terminal then it can directly be included in the first set.
* If the first symbol in the R.H.S of the production is a Non-Terminal then call the findfirst function again on that Non-Terminal. To handle these cases like Recursion is the best possible solution. Here again, if the First of the new Non-Terminal contains an epsilon then we have to move to the next symbol of the original production which can again be a Terminal or a Non-Terminal.

**Input :**

E -> TR

R -> +T R| #

T -> F Y

Y -> \*F Y | #

F -> (E) | i

**Output :**

First(E)= { (, i, }

First(R)= { +, #, }

First(T)= { (, i, }

First(Y)= { \*, #, }

First(F)= { (, i, }

-----------------------------------------------

Follow(E) = { $, ), }

Follow(R) = { $, ), }

Follow(T) = { +, $, ), }

Follow(Y) = { +, $, ), }

Follow(F) = { \*, +, $, ), }

**Program:**

|  |
| --- |
| // C program to calculate the First and  // Follow sets of a given grammar  #include<stdio.h>  #include<ctype.h>  #include<string.h>    // Functions to calculate Follow  void followfirst(char, int, int);  void follow(char c);    // Function to calculate First  void findfirst(char, int, int);    int count, n = 0;    // Stores the final result  // of the First Sets  char calc\_first[10][100];    // Stores the final result  // of the Follow Sets  char calc\_follow[10][100];  int m = 0;    // Stores the production rules  char production[10][10];  char f[10], first[10];  int k;  char ck;  int e;    int main(int argc, char \*\*argv)  {      int jm = 0;      int km = 0;      int i, choice;      char c, ch;      count = 8;        // The Input grammar      strcpy(production[0], "E=TR");      strcpy(production[1], "R=+TR");      strcpy(production[2], "R=#");      strcpy(production[3], "T=FY");      strcpy(production[4], "Y=\*FY");      strcpy(production[5], "Y=#");      strcpy(production[6], "F=(E)");      strcpy(production[7], "F=i");        int kay;      char done[count];      int ptr = -1;        // Initializing the calc\_first array      for(k = 0; k < count; k++) {          for(kay = 0; kay < 100; kay++) {              calc\_first[k][kay] = '!';          }      }      int point1 = 0, point2, xxx;        for(k = 0; k < count; k++)      {          c = production[k][0];          point2 = 0;          xxx = 0;            // Checking if First of c has          // already been calculated          for(kay = 0; kay <= ptr; kay++)              if(c == done[kay])                  xxx = 1;            if (xxx == 1)              continue;            // Function call          findfirst(c, 0, 0);          ptr += 1;            // Adding c to the calculated list          done[ptr] = c;          printf("\n First(%c) = { ", c);          calc\_first[point1][point2++] = c;            // Printing the First Sets of the grammar          for(i = 0 + jm; i < n; i++) {              int lark = 0, chk = 0;                for(lark = 0; lark < point2; lark++) {                    if (first[i] == calc\_first[point1][lark])                  {                      chk = 1;                      break;                  }              }              if(chk == 0)              {                  printf("%c, ", first[i]);                  calc\_first[point1][point2++] = first[i];              }          }          printf("}\n");          jm = n;          point1++;      }      printf("\n");      printf("-----------------------------------------------\n\n");      char donee[count];      ptr = -1;        // Initializing the calc\_follow array      for(k = 0; k < count; k++) {          for(kay = 0; kay < 100; kay++) {              calc\_follow[k][kay] = '!';          }      }      point1 = 0;      int land = 0;      for(e = 0; e < count; e++)      {          ck = production[e][0];          point2 = 0;          xxx = 0;            // Checking if Follow of ck          // has alredy been calculated          for(kay = 0; kay <= ptr; kay++)              if(ck == donee[kay])                  xxx = 1;            if (xxx == 1)              continue;          land += 1;            // Function call          follow(ck);          ptr += 1;            // Adding ck to the calculated list          donee[ptr] = ck;          printf(" Follow(%c) = { ", ck);          calc\_follow[point1][point2++] = ck;            // Printing the Follow Sets of the grammar          for(i = 0 + km; i < m; i++) {              int lark = 0, chk = 0;              for(lark = 0; lark < point2; lark++)              {                  if (f[i] == calc\_follow[point1][lark])                  {                      chk = 1;                      break;                  }              }              if(chk == 0)              {                  printf("%c, ", f[i]);                  calc\_follow[point1][point2++] = f[i];              }          }          printf(" }\n\n");          km = m;          point1++;      }  }    void follow(char c)  {      int i, j;        // Adding "$" to the follow      // set of the start symbol      if(production[0][0] == c) {          f[m++] = '$';      }      for(i = 0; i < 10; i++)      {          for(j = 2;j < 10; j++)          {              if(production[i][j] == c)              {                  if(production[i][j+1] != '\0')                  {                      // Calculate the first of the next                      // Non-Terminal in the production                      followfirst(production[i][j+1], i, (j+2));                  }                    if(production[i][j+1]=='\0' && c!=production[i][0])                  {                      // Calculate the follow of the Non-Terminal                      // in the L.H.S. of the production                      follow(production[i][0]);                  }              }          }      }  }    void findfirst(char c, int q1, int q2)  {      int j;        // The case where we      // encounter a Terminal      if(!(isupper(c))) {          first[n++] = c;      }      for(j = 0; j < count; j++)      {          if(production[j][0] == c)          {              if(production[j][2] == '#')              {                  if(production[q1][q2] == '\0')                      first[n++] = '#';                  else if(production[q1][q2] != '\0'                            && (q1 != 0 || q2 != 0))                  {                      // Recursion to calculate First of New                      // Non-Terminal we encounter after epsilon                      findfirst(production[q1][q2], q1, (q2+1));                  }                  else                      first[n++] = '#';              }              else if(!isupper(production[j][2]))              {                  first[n++] = production[j][2];              }              else              {                  // Recursion to calculate First of                  // New Non-Terminal we encounter                  // at the beginning                  findfirst(production[j][2], j, 3);              }          }      }  }    void followfirst(char c, int c1, int c2)  {      int k;        // The case where we encounter      // a Terminal      if(!(isupper(c)))          f[m++] = c;      else      {          int i = 0, j = 1;          for(i = 0; i < count; i++)          {              if(calc\_first[i][0] == c)                  break;          }            //Including the First set of the          // Non-Terminal in the Follow of          // the original query          while(calc\_first[i][j] != '!')          {              if(calc\_first[i][j] != '#')              {                  f[m++] = calc\_first[i][j];              }              else              {                  if(production[c1][c2] == '\0')                  {                      // Case where we reach the                      // end of a production                      follow(production[c1][0]);                  }                  else                  {                      // Recursion to the next symbol                      // in case we encounter a "#"                      followfirst(production[c1][c2], c1, c2+1);                  }              }              j++;          }      }  } |

Output :

First(E)= { (, i, }

First(R)= { +, #, }

First(T)= { (, i, }

First(Y)= { \*, #, }

First(F)= { (, i, }

-----------------------------------------------

Follow(E) = { $, ), }

Follow(R) = { $, ), }

Follow(T) = { +, $, ), }

Follow(Y) = { +, $, ), }

Follow(F) = { \*, +, $, ), }

**RESULT**: Thus  **first() , follow() set is implemented using c-programs and grammer rules are studied.**

**EX.NO:4**  **Removing left recursion direct as well as indirect given the set of production rule**

**AIM**: To write a C program to implement for removing left recursion direct as well as indirect given the set of production rule

### Theory:

### 1. Left Recursion-

* A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.
* A grammar containing a production having left recursion is called as a [**Left Recursive Grammar**](https://www.gatevidyalay.com/recursive-grammar-left-recursive-grammar/).

#### Example-

S → Sa / ∈

(Left Recursive Grammar)

* Left recursion is considered to be a problematic situation for Top down parsers.
* Therefore, left recursion has to be eliminated from the grammar.

**2. Right Recursion-**

A production of grammar is said to have right recursion if the rightmost variable of its RHS is same as variable of its LHS.A grammar containing a production having right recursion is called as a Right Recursive Grammar.

Example-

• S → aS / ∈

• (Right Recursive Grammar)

• Right recursion does not create any problem for the Top down parsers.

• Therefore, there is no need of eliminating right recursion from the grammar.

|  |
| --- |
| **PROGRAM:**  #include<stdio.h> |
|  | #include<string.h> |
|  | void main() { |
|  | char input[100],\*l,\*r,\*temp,tempprod[20],productions[25][50]; |
|  | int i=0,j=0,flag=0; |
|  | printf("Enter the productions: "); |
|  | scanf("%s",input); |
|  | l = strtok(input,"->"); |
|  | r = strtok(NULL,"->"); |
|  | temp = strtok(r,"|"); |
|  | while(temp) { |
|  | if(temp[0] == l[0]) { |
|  | flag = 1; |
|  | sprintf(productions[i++],"%s'->%s%s'\0",l,temp+1,l); |
|  | } |
|  | else |
|  | sprintf(productions[i++],"%s->%s%s'\0",l,temp,l); |
|  | temp = strtok(NULL,"|"); |
|  | } |
|  | sprintf(productions[i++],"%s->\356\0",l); |
|  | if(flag == 0) |
|  | printf("The given productions don't have Left Recursion"); |
|  | else |
|  | for(j=0;j<i;j++) { |
|  | printf("\n%s",productions[j]); |
|  | } |
|  | } |

Enter the productions: E->EAX|A  
E'->AXE'  
E->AE'  
E->ε  
**The 3rd production E->ε is incorrect, E'->ε is correct.**

A->Aa|b

A'->aA'  
A->bA'  
A->0

What his program has done is that now A' is the starting symbol instead of A

#include<stdio.h>

#include<string.h>

int main() {

char input[100],\*l,\*r,\*temp,tempprod[20],productions[25][50];

int i=0,j=0,flag=0;

printf("Enter the productions: ");

scanf("%s",input);

l = strtok(input,"->");

r = strtok(NULL,"->");

temp = strtok(r,"|");

while(temp) {

if(temp[0] == l[0]) {

flag = 1;

sprintf(productions[i++],"%s'->%s%s'\0",l,temp+1,l);

}

else

sprintf(productions[i++],"%s->%s%s'\0",l,temp,l);

temp = strtok(NULL,"|");

}

sprintf(productions[i++],"%s'->\u03B5",l);

if(flag == 0)

printf("The given productions don't have Left Recursion");

else

for(j=0;j<i;j++) {

printf("\n%s",productions[j]);

}

return 0;

}

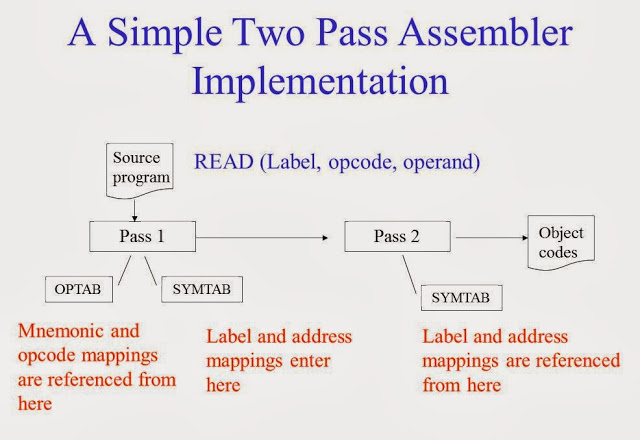
**RESULT**: Thus **Removing left recursion direct as well as indirect given the set of production rules is studied.**

**EX.NO:5**  Assemblers : 2 pass Assembler .

**AIM**: To write a C program to implement 2 pass Assembler

**Theory:**

Two-pass assemblers. An assembler is a translator, that translates an assembler programinto a conventional machine language program. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction



**ALGORITHM:**1.    Start the process.  
2.    Get the regular expression

**PROGRAM:**

1. #include<stdio.h>
2. #include<conio.h>
3. #include<string.h>
4. #include<ctype.h>
5. main()
6. {
7. FILE \*fint,\*ftab,\*flen,\*fsym;
8. int op1[10],txtlen,txtlen1,i,j=0,len;
9. char add[5],symadd[5],op[5],start[10],temp[30],line[20],label[20],mne[10],operand[10],symtab[10],opmne[10];
10. clrscr();
11. fint=fopen("input.txt","r");
12. flen=fopen("length.txt","r");
13. ftab=fopen("optab.txt","r");
14. fsym=fopen("symbol.txt","r");
15. fscanf(fint,"%s%s%s%s",add,label,mne,operand);
16. if(strcmp(mne,"START")==0)
17. {
18. strcpy(start,operand);
19. fscanf(flen,"%d",&len);
20. }
21. printf("H^%s^%s^%d\nT^00%s^",label,start,len,start);
22. fscanf(fint,"%s%s%s%s",add,label,mne,operand);
23. while(strcmp(mne,"END")!=0)
24. {
25. fscanf(ftab,"%s%s",opmne,op);
26. while(!feof(ftab))
27. {
28. if(strcmp(mne,opmne)==0)
29. {
30. fclose(ftab);
31. fscanf(fsym,"%s%s",symadd,symtab);
32. while(!feof(fsym))
33. {
34. if(strcmp(operand,symtab)==0)
35. {
36. printf("%s%s^",op,symadd);
37. break;
38. }
39. else
40. fscanf(fsym,"%s%s",symadd,symtab);
41. }
42. break;
43. }
44. else
45. fscanf(ftab,"%s%s",opmne,op);
46. }
47. if((strcmp(mne,"BYTE")==0)||(strcmp(mne,"WORD")==0))
48. {
49. if(strcmp(mne,"WORD")==0)
50. printf("0000%s^",operand);
51. else
52. {
53. len=strlen(operand);
54. for(i=2;i<len;i++)
55. {
56. printf("%d",operand[i]);
57. }
58. printf("^");
59. }
60. }
61. fscanf(fint,"%s%s%s%s",add,label,mne,operand);
62. ftab=fopen("optab.txt","r");
63. fseek(ftab,SEEK\_SET,0);
64. }
65. printf("\nE^00%s",start);
66. fclose(fint);
67. fclose(ftab);
68. fclose(fsym);
69. fclose(flen);
70. fclose(fout);
71. getch();
72. }

| **input.txt:** |
| --- |
| - | COPY | START | 1000 |
| 1000 | - | LDA | ALPHA |
| 1003 | - | ADD | ONE |
| 1006 | - | SUB | TWO |
| 1009 | - | STA | BETA |
| 1012 | ALPHA | BYTE | C'KLNCE |
| 1017 | ONE | RESB | 2 |
| 1019 | TWO | WORD | 5 |
| 1022 | BETA | RESW | 1 |
| 1025 | - | END | - |

| **optab.txt:** |
| --- |
| LDA | 00 |
| STA | 23 |
| ADD | 01 |
| SUB | 05 |

| **length.txt:25** |
| --- |

| **symbol.txt:** |
| --- |
| 1012 | ALPHA |
| 1017 | ONE |
| 1019 | TWO |
| 1022 | BETA |

**Result:** Thus 2 pass Assembler is implemented and studied using c

**EX.NO:6**  Macro processor : 2 pass Macro processor

**AIM**: To write a C program to implement 2 pass Macro processor

**Theory:**

The macro preprocessor collects any macro arguments, performs macro expansion and translation, generates labels, and then passes translated text to the statement processor. MPPL macros have the following features: Recursivity (a macro can call itself). Easy-to-read, functional syntax resembles Fortran.

Macro Processor A Macro represents a commonly used group of statements in the source programming language.

A macro instruction (macro) is a notational convenience for the programmer

• o It allows the programmer to write shorthand version of a program (module programming) The macro processor replaces each macro instruction with the corresponding

• group of source language statements (expanding)

o Normally, it performs no analysis of the text it handles.

o It does not concern the meaning of the involved statements during macro expansion. The design of a macro processor generally is machine independent!

• Two new assembler directives are used in macro definition

• o MACRO: identify the beginning of a macro definition

o MEND: identify the end of a macro definition Prototype for the macro

• o Each parameter begins with „&‟ name MACRO parameters♣ : body : MEND o Body: the statem **ALGORITHM:**1.    Start the process.  
2.    Get the regular expression ---

**PROGRAM:**

1. #include<stdio.h>
2. #include<conio.h>
3. #include<string.h>
4. #include<stdlib.h>
5. void main()
6. {
7. FILE \*f1,\*f2,\*f3,\*f4,\*f5;
8. int i,len;
9. char mne[20],opnd[20],la[20],name[20],mne1[20],opnd1[20],arg[20];
10. clrscr();
11. f1=fopen("minp2.txt","r");
12. f2=fopen("ntab2.txt","r");
13. f3=fopen("dtab2.txt","r");
14. f4=fopen("atab2.txt","w+");
15. f5=fopen("op2.txt","w");
16. fscanf(f1,"%s%s%s",la,mne,opnd);
17. while(strcmp(mne,"END")!=0)
18. {
19. if(strcmp(mne,"MACRO")==0)
20. {
21. fscanf(f1,"%s%s%s",la,mne,opnd);
22. while(strcmp(mne,"MEND")!=0)
23. fscanf(f1,"%s%s%s",la,mne,opnd);
24. }
25. else
26. {
27. fscanf(f2,"%s",name);
28. if(strcmp(mne,name)==0)
29. {
30. len=strlen(opnd);
31. for(i=0;i<len;i++)
32. {
33. if(opnd[i]!=',')
34. fprintf(f4,"%c",opnd[i]);
35. else
36. fprintf(f4,"\n");
37. }
38. fseek(f2,SEEK\_SET,0);
39. fseek(f4,SEEK\_SET,0);
40. fscanf(f3,"%s%s",mne1,opnd1);
41. fprintf(f5,".\t%s\t%s\n",mne1,opnd);
42. fscanf(f3,"%s%s",mne1,opnd1);
43. while(strcmp(mne1,"MEND")!=0)
44. {
45. if((opnd1[0]=='&'))
46. {
47. fscanf(f4,"%s",arg);
48. fprintf(f5,"-\t%s\t%s\n",mne1,arg);
49. }
50. else
51. fprintf(f5,"-\t%s\t%s\n",mne1,opnd1);
52. fscanf(f3,"%s%s",mne1,opnd1);
53. }
54. }
55. else
56. fprintf(f5,"%s\t%s\t%s\n",la,mne,opnd);
57. }
58. fscanf(f1,"%s%s%s",la,mne,opnd);
59. }
60. fprintf(f5,"%s\t%s\t%s\n",la,mne,opnd);
61. fclose(f1);
62. fclose(f2);
63. fclose(f3);
64. fclose(f4);
65. fclose(f5);
66. printf("pass2");
67. getch();
68. }

Input files:

| **minp2.txt** |
| --- |
| EX1 | MACRO | &A,&B |
| - | LDA | &A |
| - | STA | &B |
| - | MEND | - |
| SAMPLE | START | 1000 |
| - | EX1 | N1,N2 |
| N1 | RESW | 1 |
| N2 | RESW | 1 |
| - | END | - |

| **dtab2.txt** |
| --- |
| EX1 | &A,&B |
| LDA | &A |
| STA | &B |
| MEND |  |

| **ntab2.txt** |
| --- |
| EX1 |

Output files:

| **atab2.txt** |
| --- |
| N1 |
| N2 |

| **op2.txt** |
| --- |
| SAMPLE | START | 1000 |
| . | EX1 | N1,N2 |
| - | LDA | N1 |
| - | STA | N2 |
| N1 | RESW | 1 |
| N2 | RESW | 1 |
| - | END | - |

**Result:** Thus 2 pass macro-processor is implemented and studied using c

**EX.NO:7**  Syntax Analysis : ( any 1 of LL(1) , LR(0) , LR(1) , LALR(1) , operator precedence parser.

**AIM**: To write a C program to implement & construction of operator precedence parse table

**THEORY:**

An operator-precedence parser is a simple [shift-reduce parser](https://en.wikipedia.org/wiki/Shift-reduce_parser) that is capable of parsing a subset of [LR(1)](https://en.wikipedia.org/wiki/LR_parser) grammars. More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive [nonterminals](https://en.wikipedia.org/wiki/Nonterminal) and epsilon never appear in the right-hand side of any rule.

Operator-precedence parsers are not used often in practice; however they do have some properties that make them useful within a larger design. First, they are simple enough to write by hand, which is not generally the case with more sophisticated right shift-reduce parsers. Second, they can be written to consult an operator table at [run time](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)), which makes them suitable for languages that can add to or change their operators while parsing. (An example is [Haskell](https://en.wikipedia.org/wiki/Haskell_(programming_language)), which allows user-defined infix operators with custom associativity and precedence; consequentially, an operator-precedence parser must be run on the program after parsing of all referenced modules.

**ALGORITHM:**1.    Start the process.  
2.    Get the regular expression.  
3.    Get the no of terminals including $.  
4.    For identifier ‘i’ and operators ‘+’,’\*’ are used.  
5.    Enter the terminals in row and column wise for each entry.  
6.    For each identifier read make an entry in operator precedence parse table.  
7.    For each symbol mention its type, value, relocatable location and size.  
8.    Finally update the symbols in priority basis to the table.  
  
**PROGRAM:**#include<stdio.h>  
#include<conio.h>  
  
void main( )  
{  
    int i,j,n,ib=0,top=0,num[10],k,r,t,l,x,y,z,g=0,li;  
    char tab[10][10],s[10],stack[20],inter[10],prod[10][10];  
    clrscr( );  
    printf(“\n Enter the input expression:\n”);  
    scanf(“%s”,s);  
    printf(“\n Enter the number of terminals INC $:\n”);  
    scanf(“%d”,&n);  
    for(i=0;i<=n;i++)  
    for(j=0;j<=n;j++)  
    tab[i][j]=’ ‘;  
printf({“\n Give The entries in first row and column corresponding to terminal:\n”);  
num[0]=’\0’;  
for(i=1;i<=n;i++)  
{  
     scanf(“%s”,&tab[0][i]);  
    num[i]=tab[0][i];  
      
    tab[i][0]=tab[0][i];  
}  
printf(“\n Enter the precedence table row wise:\n”);  
for(i=1;i<=n;i++)  
{  
    for(j=1;j<=n;j++)  
    {  
        scanf(“%s”,&tab[i][j]);  
    }  
}  
for(i=0;i<=n;i++)  
{  
  
  
    for(j=0;j<=n;j++)  
    {  
        printf(“%c\t”,tab[i][j]);  
    }  
    printf(“\n\n”);  
}  
stack[top]= ‘$’;  
l=strlen(s);  
s[l]=’$’;  
s[l+1]=’\0’;  
printf(“\nStack \t Input buffer \t Popped Elements:\n”);  
printf(“$\t%s\n”,s);  
for(i=0;i<=l;i++)  
{  
    for(k=1;k<=n;k++)  
    {  
        if(stack[top]= =num[k])  
        x=k;  
        if(s[i] = =num[k])  
        y=k;  
    }  
    if(tab[x][y]= = ‘=’ || tab[x][y] = = ‘<’)  
    {  
        stack[top+1]=s[i];  
          
        top++;  
        ib++;  
      
    }  
    else if(tab[x][y]= = ‘>’)  
    {  
        inter[g]=stack[top];  
        g++;  
        stack[top]=s[i];  
        ib++;  
    }  
    for(r=0;r<=top;r++)  
    printf(“%c”,stack[r]);  
        printf(“\t”);  
    for(t=ib;t<=l;t++)  
    printf(“%c”,s[t]);  
    printf(“\t\t”);  
    if(tab[x][y] = = ‘>’)  
    {  
        printf(“%c”, inter[g-1]);  
    }  
    printf(“\n”);  
}  
top--;  
while(stack[top]!=’$’)  
{  
    inter[g]=stack[top];  
    g++;  
    top--;  
    for(r=0;r<=top;r++)  
    printf(“%c”,stack[r]);  
    printf(“\t\t\t”);  
    printf(“%c”,inter[g-1]);  
    printf(“\n”);  
}  
getch( );}  
  
**OUTPUT:**      
Enter the input  expression:  
    i+i\*i  
      
    Enter the number of terminals including $:  
    4  
Give the entries in first row and column corresponding to terminal:  
    i  
    +  
    \*  
    $  
      
    Enter the precedence table row wise:  
    -  
    >  
    >  
    >  
    <  
    >  
    <  
    >  
    <  
    >  
    >  
    >  
    <  
    <  
    <  
    -  
   Operator Precedence Table:  
  
        i    +    \*     $  
      
    i    -    >    >    >  
      
    +    <    >    <    >  
      
    \*    <    >    >    >  
  
    $    <    <    <    -  
  
STACK         INPUT BUFFER        POPPED ELEMENTS:  
  
$            i+i\*i$  
  
$i            +i\*i$  
  
$+            i\*i$                i  
  
$+i            \*i$          
  
$+\*            i$                i  
  
$+\*i            $  
  
$+\*                            i  
  
$+                            \*  
$                            +

**RESULT:**       Thus the operator precedence parse table program was executed and verified

**EX.NO:8** Creation of your library in Linux environment and using it.

**AIM**: Create of your library in Linux environment and using it.

**THEORY:**

**1.Function**: A function is a collection of declarations and statements that carries out a specific action and/or returns a value.  Previously defined functions that have related functionality or are commonly used (e.g. math or graphics routines) are stored in object code format in library files. Object code format is a special file format that is generated as an intermediate step when an executable program is produced. Like executable files, object code files are also not displayed to the screen or printed. Functions stored in library files are often called library functions or runtime library routines.

The standard location for library file in most Unix system is the directory /usr/lib and /usr/local/lib...

**2.Libraries name convention**: Be convention, the three-letter prefix for a library file is lib and the file extension for a static library is .a. For example, I will show you how you can make a library with name: libmydemo.a later.

**3.See the content of a library such as /use/lib/libc.a :**

**4.  Generation our own library.**

**PROGRAM:**

The ar unility can also be used to create a library. For example, we have 2 functions: ascii which is in ascii.c  and change\_case which is in change\_case.c files.

//File: ascii.c

  char \* ascii(int start, int finish)

   {

     char \*b= new char(finish-start+1);

     for (int i= start; i<=finish; ++i)

         b[i-start]=char(i);

          return b;

    }

//File: change\_case.c

  #include<ctype.h>

  char\* change\_case(char \*s)

  {

   char \*t = &s[0];

       while(\*t)

          {

          if(isalpha(\*t))

         \*t+=islower(\*t)?-32:32;

         ++t;

          }

    return s;

  }

 $ g++ -c change\_case.c

 $ g++ -c ascii.c

 $ ar cr libmydemo.a ascii.o change\_case.o

What I did above is the compile the code to object code, and the object code added to the archive with the utility ar.

 The prototypes for the functions in the mydemo library are placed in a corresponding header file called mydemo.h Be sure to use preprocessor directives in that file to prevent it from being inadvertently included more than once.

//File mydemo.h

  #ifndef MYDEMO\_H

  #define MYDEMO\_H

   char\* ascii(int, int);

   char\* change\_case(char\*);

#endif

Well, How to use the library I created?

Here is an example:

 File: main.c

#include<iostream>

#include"mydemo.h"

using namespace std;

main()

{

int start, stop;

char b[20];

cout<<"enter the start and stop value for string:"<<endl;

cin>>start>>stop;

cout<<"create string..."<<ascii(start, stop)<<endl;

cin.ignore(80,'\n');

cout<<"enter a string :";

cin.getline(b,20);

cout<<"coverted string: "<<change\_case(b)<<endl;

return 0;

}

Next step is to compile and link it:

 $ g++ -o main mani.c -L. -lmydemo

Here, "-L." is to use to tell the system that when the compiler searches for library files it should also include the current directory. The name of the library is passed using the -lcommand option, you call the library without the lib prefix and the .a extension..

Ok, now let's run it?

$ main

enter the start and stop value for string:

56 68

create string... 89:,0<1321@ABCD

enter a string : Hello, World.

coverted string: hELLO, wORLD.

aha, wonderful!

//Huican ping notes:

**RESULT**: Thus library in Linux environment is createdand using it.

**EX.NO:9**  Implementation Of Intermediate Code Generation  
  
**AIM**:  To write a C program to implement the intermediate code for the given set of input expressions.

**THEORY:**

Intermediate Code Generation. In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).  
  
**ALGORITHM:**  
1.    Start the program.  
2.    Get the input expression from the user.  
3.    Check the expressions for its validation.  
4.    If it is invalid return the error message.  
5.    Otherwise, for each computation store the result in the three – address statement (store it in temporary variable say t1, t2, etc.,) .  
6.    Assign the final temporary value to the variable in which the result has to be stored.  
7.    Stop the program.  
          
PROGRAM:   
#include<stdio.h>  
#include<ctype.h>  
#include<stdlib.h>  
#include<conio.h>  
#include<string.h>  
void small();  
void dove(int );  
int p[5]={0,1,2,3,4},c=1,i,k,l,m,pi;  
char sw[5]={'=','-','+','/','\*'},j[20],a[5],b[5],ch[2];  
void main()  
{  
      clrscr();  
      printf("Enter the expression:");  
      scanf("%s",j);  
      printf("\n\n\tThe Intermediate code is:\n");  
      small();  
}  
void dove(int i)  
{   
a[0]='\0';b[0]='\0';  
    
I    f(!isdigit(j[i+2]) && !isdigit(j[i-2]))  
       {  
             a[0]=j[i-1];  
             b[0]=j[i+1];  
       }  
       if(isdigit(j[i+2]))  
       {  
        a[0]=j[i-1];  
        b[0]='t';  
        b[1]=j[i+2];  
       }  
       if(isdigit(j[i-2]))  
       {    b[0]=j[i+1];  
        a[0]='t';  
        a[1]=j[i-2];  
        b[1]='\0';        }  
       if(isdigit(j[i+2]) && isdigit(j[i-2]))  
       {    a[0]='t';  
        b[0]='t';  
        a[1]=j[i-2];  
        b[1]=j[i+2];  
        itoa(c,ch,10);  
        j[i+2]=j[i-2]=ch[0];       }  
      if(j[i]=='\*')  
            printf("\tt%d=%s\*%s\n",c,a,b);  
      if(j[i]=='/')  
            printf("\tt%d=%s/%s\n",c,a,b);  
      if(j[i]=='+')  
            printf("\tt%d=%s+%s\n",c,a,b);  
      if(j[i]=='-')  
            printf("\tt%d=%s-%s\n",c,a,b);  
      if(j[i]=='=')  
            printf("\t%c=t%d",j[i-1],--c);  
      itoa(c,ch,10);  
      j[i]=ch[0];  
      c++;  
      small();  
}  
  
void small()  
{      pi=0;l=0;  
      for(i=0;i<strlen(j);i++)  
      {    for(m=0;m<5;m++)  
                   if(j[i]==sw[m])  
                  if(pi<=p[m])  
                  {  
                        pi=p[m];  
                        l=1;  
                        k=i;  
                  }      }  
      if(l==1)  
            dove(k);  
      else  
      {  
            getch();  
            exit (0);  
      } }          
  
**OUTPUT:**Enter the expression:  
a=b+c\*d-e/f  
The Intermediate code is:  
        t1=c\*d  
        t2=e/f  
        t3=b+t1  
        t4=t1-t2  
        a=t4               
**RESULT:**       Thus the intermediate code for the given set of input expressions is implemented.

**EX.NO:10**   Implementation of Code Optimization Techniques  
**AIM**: To write a C program to implement the code optimization techniques  
**THEORY**:

Optimization is a program transformation technique, which tries to improve thecode by making it consume less resources (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-levelprogramming codes. **ALGORITHM:**  
    The code generation algorithm takes as input a sequence of three – address statements constituting a basic block. For each three – address statement of the form  x := y op z we perform the following actions:  
    
1.    Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L will usually be a register, but it could also be a memory location. We shall describe getreg shortly.  
, L to place a copy of y in L.′ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y′, (one of) the current location(s) of y. prefer the register for y′2.    Consult the address descriptor for y to determine y  
 is a current location of z. Again, prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.′, L where z′3.    Generate the instruction OP z  
4.    If the current values of y and/or z have no next users, are not live on exit from the block, and are in register descriptor to indicate that, after execution of x := y op z, those registers no longer will contain y and/or z, respectively.  
  
**PROGRAM:**  
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
struct op  
{  
char l;  
char r[20];  
}op[10],pr[10];  
  
void main()  
{  
int a,i,k,j,n,z=0,m,q;  
  
char \*p,\*l;  
char temp,t;  
char \*tem;  
clrscr();  
printf("enter no of values");  
scanf("%d",&n);  
for(i=0;i<n;i++)  
{  
printf("left\t");  
op[i].l=getche();  
printf("right:\t");  
scanf("%s",op[i].r);  
}  
printf("intermediate Code\n") ;  
for(i=0;i<n;i++)  
{  
printf("%c=",op[i].l);  
printf("%s\n",op[i].r);  
}  
for(i=0;i<n-1;i++)  
{  
temp=op[i].l;  
for(j=0;j<n;j++)  
{  
p=strchr(op[j].r,temp);  
if(p)  
{  
pr[z].l=op[i].l;  
strcpy(pr[z].r,op[i].r);  
z++ ;  
  
}} }  
pr[z].l=op[n-1].l;  
strcpy(pr[z].r,op[n-1].r);  
z++;  
printf("\nafter dead code elimination\n");  
for(k=0;k<z;k++)  
{  
  
printf("%c\t=",pr[k].l);  
printf("%s\n",pr[k].r);  
}  
  
//sub expression elimination  
for(m=0;m<z;m++)  
{  
tem=pr[m].r;  
for(j=m+1;j<z;j++)  
{  
p=strstr(tem,pr[j].r);  
if(p)  
{  
t=pr[j].l;  
pr[j].l=pr[m].l     ;  
for(i=0;i<z;i++)  
{  
l=strchr(pr[i].r,t) ;  
if(l)  
{  
a=l-pr[i].r;  
//printf("pos: %d",a);  
pr[i].r[a]=pr[m].l;  
}}}}}  
printf("eliminate common expression\n");  
for(i=0;i<z;i++)  
{  
printf("%c\t=",pr[i].l);  
printf("%s\n",pr[i].r);  
}  
// duplicate production elimination  
  
for(i=0;i<z;i++)  
{  
for(j=i+1;j<z;j++)  
{  
q=strcmp(pr[i].r,pr[j].r);  
if((pr[i].l==pr[j].l)&&!q)  
  
{  
    pr[i].l='\0';  
    strcpy(pr[i].r,'\0');  
 }}  
}  
printf("optimized code");  
for(i=0;i<z;i++)  
{  
if(pr[i].l!='\0')  
{  
printf("%c=",pr[i].l);  
printf("%s\n",pr[i].r);  
}  
}  
getch();  
}  
  
**OUTPUT:**  
enter no of values 5  
left    aright:  9  
left    bright:  c+d  
left    eright:  c+d  
left    fright:  b+e  
left    rright:  f  
intermediate Code  
a=9  
b=c+d  
e=c+d  
f=b+e  
r=f  
  
after dead code elimination  
b       =c+d  
e       =c+d  
f       =b+e  
r       =f  
eliminate common expression  
b       =c+d  
b       =c+d  
f       =b+b  
r       =f  
optimized codeb=c+d  
f=b+b  
r=f  
                        
**RESULT:**  
Thus the above program is compiled and executed successfully and output is verified.

**EX.NO:11** Study ld command in Unix/Linux.  
**AIM**: Study and implementation of ld command in Unix/Linux  
**THEORY**:

The **ld command**, also called the linkage editor or binder, combines object files, archives, and import files into one output object file, resolving external references. It produces an executable object file that can be run. ... By default, the **ld command**creates and places its output in the a.out file.

**ALGORITHM**  
Study ld command in Unix/Linux.

|  |  |
| --- | --- |
| @file | Read command-line options from file. The options read are inserted in place of the original @file option. If file does not exist, or cannot be read, then the option will be treated literally, and not removed.  Options in file are separated by [whitespace](https://www.computerhope.com/jargon/w/whitspac.htm). A whitespace character may be included in an option by surrounding the entire option in either single or double quotes. Any character (including a backslash) may be included by prefixing the character to be included with a backslash. The file may itself contain additional @file options; any such options will be processed [recursively](https://www.computerhope.com/jargon/r/recursive.htm). |
| -a keyword | This option is supported for [HP-UX](https://www.computerhope.com/jargon/h/hp-ux.htm) compatibility. The keywordargument must be one of the [strings](https://www.computerhope.com/jargon/s/string.htm) archive, shared, or default. -aarchive is functionally equivalent to -Bstatic, and the other two keywords are functionally equivalent to -Bdynamic. This option may be used any number of times. |
| --audit AUDITLIB | Adds AUDITLIB to the "DT\_AUDIT" entry of the dynamic section. AUDITLIB is not checked for existence, nor will it use the DT\_SONAME specified in the library. If specified multiple times "DT\_AUDIT" will contain a colon separated list of audit interfaces to use. If the linker finds an object with an audit entry while searching for shared libraries, it will add a corresponding "DT\_DEPAUDIT" entry in the output file. This option is only meaningful on ELF platforms supporting the "rtld-audit" interface. |
| -A architecture  --architecture=architecture | In the current release of ld, this option is useful only for the [Intel](https://www.computerhope.com/comp/intel.htm) 960 family of [architectures](https://www.computerhope.com/jargon/a/architec.htm). In that ld configuration, the architecture argument identifies the particular architecture in the 960 family, enabling some safeguards and modifying the archive-library search path.  Future releases of ld may support similar functionality for other architecture families. |
| -b input-format  --format=input-format | ld may be configured to support more than one kind of object file. If your ld is configured this way, you can use the -b option to specify the binary format for input object files that follow this option on the command line. Even when ld is configured to support alternative object formats, you don't usually need to specify this, as ld should be configured to expect as a default input format the most usual format on each machine. The input-format is a text string, the name of a particular format supported by the BFD libraries. You can list the available binary formats with objdump -i.  You may want to use this option if you are linking files with an unusual binary format. You can also use -b to switch formats explicitly (when linking object files of different formats), by including -b input-format before each group of object files in a particular format.  The default format is taken from the [environment variable](https://www.computerhope.com/jargon/e/envivari.htm)"GNUTARGET".  You can also define the input format from a script, using the command "TARGET"; |
| -c MRI-commandfile  --mri-script=MRI-commandfile | For compatibility with linkers produced by MRI, ld accepts script files written in an alternate, restricted command language, described in the MRI Compatible Script Files section of GNU ld documentation. Introduce MRI script files with the option -c; use the -T option to run linker scripts written in the general-purpose ld scripting language. If MRI-cmdfile does not exist, ld looks for it in the directories specified by any -L options. |
| -d  -dc  -dp | These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file is specified (with -r). The script command "FORCE\_COMMON\_ALLOCATION" has the same effect. |
| --depaudit AUDITLIB  -P AUDITLIB | Adds AUDITLIB to the "DT\_DEPAUDIT" entry of the dynamic section. AUDITLIB is not checked for existence, nor will it use the DT\_SONAME specified in the library. If specified multiple times "DT\_DEPAUDIT" will contain a colon separated list of audit interfaces to use. This option is only meaningful on ELF platforms supporting the "rtld-audit" interface. The -P option is provided for Solaris compatibility. |
| -e entry  --entry=entry | Use entry as the explicit symbol for beginning execution of your program, rather than the default entry point. If there is no symbol named entry, the linker will try to parse entry as a number, and use that as the entry address (the number will be interpreted in [base 10](https://www.computerhope.com/jargon/d/decimal.htm); you may use a leading 0x for base 16, or a leading 0 for base 8). |
| --exclude-libs lib,lib,... | Specifies a list of archive libraries from which symbols should not be automatically exported. The library names may be [delimited](https://www.computerhope.com/jargon/d/delimite.htm) by commas or colons. Specifying "--exclude-libs ALL" excludes symbols in all archive libraries from automatic export. This option is available only for the i386 PE targeted port of the linker and for ELF targeted ports. For i386 PE, symbols explicitly listed in a .def file are still exported, regardless of this option. For ELF targeted ports, symbols affected by this option will be treated as hidden. |
| --exclude-modules-for-implib module,module,... | Specifies a list of object files or archive members, from which symbols should not be automatically exported, but which should be copied wholesale into the import library being generated during the link. The module names may be delimited by commas or colons, and must match exactly the filenames used by ld to open the files; for archive members, this is the member name, but for object files the name listed must include and match precisely any path used to specify the input file on the linker's command-line. This option is available only for the i386 PE targeted port of the linker. Symbols explicitly listed in a .def file are still exported, regardless of this option. |
| -E  --export-dynamic  --no-export-dynamic | When creating a dynamically linked executable, using the -E option or the --export-dynamic option causes the linker to add all symbols to the dynamic symbol table. The dynamic symbol table is the set of symbols that are visible from dynamic objects at run time.  If you do not use either of these options (or use the --no-export-dynamic option to restore the default behavior), the dynamic symbol table will normally contain only those symbols that are referenced by some dynamic object mentioned in the link.  If you use "dlopen" to load a dynamic object which needs to refer back to the symbols defined by the program, rather than some other dynamic object, then you will probably need to use this option when linking the program itself.  You can also use the dynamic list to control what symbols should be added to the dynamic symbol table if the output format supports it. See the description of --dynamic-list.  Note that this option is specific to ELF targeted ports. PE targets support a similar function to export all symbols from a [DLL](https://www.computerhope.com/jargon/d/dll.htm) or [EXE](https://www.computerhope.com/jargon/e/execfile.htm); see the description of --export-all-symbols below. |
| -EB | Link big-[endian](https://www.computerhope.com/jargon/e/endian.htm) objects. This affects the default output format. |
| -EL | Link little-endian objects. This affects the default output format. |
| -f name  --auxiliary=name | When creating an ELF shared object, set the internal DT\_AUXILIARY field to the specified name. This tells the dynamic linker that the symbol table of the shared object should be used as an auxiliary filter on the symbol table of the shared object name.  If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the DT\_AUXILIARY field. If the dynamic linker resolves any symbols from the filter object, it will first check whether there is a definition in the shared object name. If there is one, it will be used instead of the definition in the filter object. The shared object name need not exist. Thus the shared object name may be used to provide an alternative implementation of certain functions, perhaps for debugging or for machine specific performance.  This option may be specified more than once. The DT\_AUXILIARY entries will be created in the order in which they appear on the command line. |
| -F name  --filter=name | When creating an ELF shared object, set the internal DT\_FILTER field to the specified name. This tells the dynamic linker that the symbol table of the shared object that is being created should be used as a filter on the symbol table of the shared object name.  If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the DT\_FILTER field. The dynamic linker will resolve symbols according to the symbol table of the filter object as usual, but it will actually link to the definitions found in the shared object name. Thus the filter object can be used to select a subset of the symbols provided by the object name.  Some older linkers used the -F option throughout a compilation toolchain for specifying object-file format for both input and output object files. The GNU linker uses other mechanisms for this purpose: the -b, --format, --oformat options, the "TARGET" command in linker scripts, and the "GNUTARGET" environment variable. The GNU linker will ignore the -F option when not creating an ELF shared object. |
| -fini=name | When creating an ELF executable or shared object, call NAME when the executable or shared object is unloaded, by setting DT\_FINI to the address of the function. By default, the linker uses "\_fini" as the function to call. |
| -g | Ignored. Provided for compatibility with other tools. |
| -G value  --gpsize=value | Set the maximum size of objects to be optimized using the GP register to size. This is only meaningful for object file formats such as MIPS ECOFF which supports putting large and small objects into different sections. This is ignored for other object file formats. |
| -h name  -soname=name | When creating an ELF shared object, set the internal DT\_SONAME field to the specified name. When an executable is linked with a shared object which has a DT\_SONAME field, then when the executable is run the dynamic linker will attempt to load the shared object specified by the DT\_SONAME field rather than the using the file name given to the linker. |
| -i | Perform an incremental link (same as option -r). |
| -init=name | When creating an ELF executable or shared object, call NAME when the executable or shared object is loaded, by setting DT\_INIT to the address of the function. By default, the linker uses "\_init" as the function to call. |
| -l namespec  --library=namespec | Add the archive or object file specified by namespec to the list of files to link. This option may be used any number of times. If namespec is of the form :filename, ld will search the library path for a file called filename, otherwise it will search the library path for a file called libnamespec.a.  On systems which support shared libraries, ld may also search for files other than libnamespec.a. Specifically, on ELF and [SunOS](https://www.computerhope.com/comp/sun.htm)systems, ld will search a directory for a library called libnamespec.sobefore searching for one called libnamespec.a. By convention, a ".so" extension indicates a shared library. Note that this behavior does not apply to :filename, which always specifies a file called filename.  The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol which was undefined in some object which appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again.  See the -( option for a way to force the linker to search archives multiple times.  You may list the same archive multiple times on the command line.  This type of archive searching is standard for Unix linkers. However, if you are using ld on [AIX](https://www.computerhope.com/jargon/a/aix.htm), note that it is different from the behaviour of the AIX linker. |
| -L searchdir  --library-path=searchdir | Add path searchdir to the list of paths that ld will search for archive libraries and ld control scripts. You may use this option any number of times. The directories are searched in the order in which they are specified on the command line. Directories specified on the command line are searched before the default directories. All -L options apply to all -l options, regardless of the order in which the options appear. -Loptions do not affect how ld searches for a linker script unless -Toption is specified.  If searchdir begins with "=", then the "=" will be replaced by the sysroot prefix, a path specified when the linker is configured.  The default set of paths searched (without being specified with -L) depends on which emulation mode ld is using, and in some cases also on how it was configured at build time.  The paths can also be specified in a link script with the "SEARCH\_DIR" command. Directories specified this way are searched at the point in which the linker script appears in the command line. |
| -m emulation | Emulate the emulation linker. You can list the available emulations with the --verbose or -V options.  If the -m option is not used, the emulation is taken from the "LDEMULATION" environment variable, if that is defined.  Otherwise, the default emulation depends upon how the linker was configured. |
| -M  --print-map | Print a link map to the standard output. A link map provides information about the link, including the following:   * Where object files are mapped into memory. * How common symbols are allocated. * All archive members included in the link, with a mention of the symbol that caused the archive member. * The values assigned to symbols.   Note: symbols whose values are computed by an [expression](https://www.computerhope.com/jargon/e/expressi.htm) which involves a reference to a previous value of the same symbol may not have correct result displayed in the link map. This is because the linker discards intermediate results and only retains the final value of an expression. Under such circumstances the linker will display the final value enclosed by square brackets. Thus for example a linker script containing:  foo = 1  foo = foo \* 4  foo = foo + 8  will produce the following output in the link map if the -M option is used:  0x00000001 foo = 0x1  [0x0000000c] foo = (foo \* 0x4)  [0x0000000c] foo = (foo + 0x8) |
| -n  --nmagic | Turn off page alignment of sections, and disable linking against shared libraries. If the output format supports Unix style magic numbers, mark the output as "NMAGIC". |
| -N  --omagic | Set the text and data sections to be readable and writable. Also, do not page-align the data segment, and disable linking against shared libraries. If the output format supports Unix style magic numbers, mark the output as "OMAGIC". Note: Although a writable text section is allowed for PE-COFF targets, it does not conform to the format specification published by [Microsoft](https://www.computerhope.com/comp/msoft.htm). |
| --no-omagic | This option negates most of the effects of the -N option. It sets the text section to be read-only, and forces the data segment to be page-aligned. Note - this option does not enable linking against shared libraries. Use -Bdynamic for this. |
| -o output  --output=output | Use output as the name for the program produced by ld; if this option is not specified, the name a.out is used by default. The script command "OUTPUT" can also specify the output file name. |
| -O level | If level is a numeric value greater than zero ld optimizes the output. This might take significantly longer and therefore probably should only be enabled for the final binary. At the moment this option only affects ELF shared library generation. Future releases of the linker may make more use of this option. Also, currently there is no difference in the linker's behaviour for different non-zero values of this option. Again this may change with future releases. |
| -q  --emit-relocs | Leave relocation sections and contents in fully linked executables. Post link analysis and optimization tools may need this information to perform correct modifications of executables. This results in larger executables.  This option is currently only supported on ELF platforms. |
| --force-dynamic | Force the output file to have dynamic sections. This option is specific to VxWorks targets. |
| -r  --relocatable | Generate relocatable output: i.e., generate an output file that can in turn serve as input to ld. This is often called partial linking. As a side effect, in environments that support standard Unix magic numbers, this option also sets the output file's magic number to "OMAGIC". If this option is not specified, an absolute file is produced. When linking C++ programs, this option will not resolve references to constructors; to do that, use -Ur.  When an input file does not have the same format as the output file, partial linking is only supported if that input file does not contain any relocations. Different output formats can have further restrictions; for example some "a.out"-based formats do not support partial linking with input files in other formats at all.  This option does the same thing as -i. |
| -R filename  --just-symbols=filename | Read symbol names and their addresses from filename, but do not relocate it or include it in the output. This allows your output file to refer symbolically to absolute locations of memory defined in other programs. You may use this option more than once.  For compatibility with other ELF linkers, if the -R option is followed by a directory name, rather than a file name, it is treated as the -rpathoption. |
| -s  --strip-all | Omit all symbol information from the output file. |
| -S  --strip-debug | Omit debugger symbol information (but not all symbols) from the output file. |
| -t  --trace | Print the names of the input files as ld processes them. |
| -T scriptfile  --script=scriptfile | Use scriptfile as the linker script. This script replaces ld's default linker script (rather than adding to it), so commandfile must specify everything necessary to describe the output file. If scriptfile does not exist in the current directory, ld looks for it in the directories specified by any preceding -L options. Multiple -T options accumulate. |
| -dT scriptfile  --default-script=scriptfile | Use scriptfile as the default linker script.  This option is similar to the --script option except that processing of the script is delayed until after the rest of the command line has been processed. This allows options placed after the --default-scriptoption on the command line to affect the behaviour of the linker script, which can be important when the linker command line cannot be directly controlled by the user. (e.g., because the command line is being constructed by another tool, such as [gcc](https://www.computerhope.com/jargon/g/gcc.htm)). |
| -u symbol  --undefined=symbol | Force symbol to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. -u may be repeated with different option arguments to enter additional undefined symbols. This option is equivalent to the "EXTERN" linker script command. |
| -Ur | For anything other than C++ programs, this option is equivalent to -r: it generates relocatable output; i.e., an output file that can in turn serve as input to ld. When linking C++ programs, -Ur does resolve references to constructors, unlike -r. It does not work to use -Ur on files that were themselves linked with -Ur; once the constructor table has been built, it cannot be added to. Use -Ur only for the last partial link, and -r for the others. |
| --unique[=SECTION] | Creates a separate output section for every input section matching SECTION, or if the optional wildcard SECTION argument is missing, for every orphan input section. An orphan section is one not specifically mentioned in a linker script. You may use this option multiple times on the command line; It prevents the normal merging of input sections with the same name, overriding output section assignments in a linker script. |
| -v  --version  -V | Display the version number for ld. The -V option also lists the supported emulations. |
| -x  --discard-all | Delete all local symbols. |
| -X  --discard-locals | Delete all temporary local symbols. These symbols start with system-specific local label prefixes, typically .L for ELF systems or L for traditional a.out systems. |
| -y symbol --trace-symbol=symbol | Print the name of each linked file in which symbol appears. This option may be given any number of times. On many systems it is necessary to prepend an underscore.  This option is useful when you have an undefined symbol in your link but don't know where the reference is coming from. |
| -Y path | Add path to the default library search path. This option exists for [Solaris](https://www.computerhope.com/jargon/s/solaris.htm) compatibility. |
| -z keyword | The recognized keywords are:   |  |  | | --- | --- | | combreloc | Combines multiple reloc sections and sorts them to make dynamic symbol lookup caching possible. | | defs | Disallows undefined symbols in object files. Undefined symbols in shared libraries are still allowed. | | execstack | Marks the object as requiring executable stack. | | initfirst | This option is only meaningful when building a shared object. It marks the object so that its runtime initialization will occur before the runtime initialization of any other objects brought into the process at the same time. Similarly the runtime finalization of the object will occur after the runtime finalization of any other objects. | | interpose | Marks the object that its symbol table interposes before all symbols but the primary executable. | | lazy | When generating an executable or shared library, mark it to tell the dynamic linker to defer function call resolution to the point when the function is called (lazy binding), rather than at load time. Lazy binding is the default. | | loadfltr | Marks the object that its filters be processed immediately at runtime. | | muldefs | Allows multiple definitions. | | nocombreloc | Disables multiple reloc sections combining. | | nocopyreloc | Disables production of copy relocs. | | nodefaultlib | Marks the object that the search for dependencies of this object will ignore any default library search paths. | | nodelete | Marks the object shouldn't be unloaded at runtime. | | nodlopen | Marks the object not available to "dlopen". | | nodump | Marks the object can not be dumped by "dldump". | | noexecstack | Marks the object as not requiring executable stack. | | norelro | Don't create an ELF "PT\_GNU\_RELRO" segment header in the object. | | now | When generating an executable or shared library, mark it to tell the dynamic linker to resolve all symbols when the program is started, or when the shared library is linked to using dlopen, instead of deferring function call resolution to the point when the function is first called. | | origin | Marks the object may contain $ORIGIN. | | relro | Create an ELF "PT\_GNU\_RELRO" segment header in the object. | | max-page-size=value | Set the emulation maximum page size to value. | | common-page-size=value | Set the emulation common page size to value. |   Other keywords are ignored for Solaris compatibility. |
| -( archives -)  --start-group archives --end-group | The archives should be a list of archive files. They may be either explicit file names, or -l options.  The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they all be searched repeatedly until all possible references are resolved.  Using this option has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more archives. |
| --accept-unknown-input-arch  --no-accept-unknown-input-arch | Tells the linker to accept input files whose architecture cannot be recognised. The assumption is that the user knows what they are doing and deliberately wants to link in these unknown input files. This was the default behaviour of the linker, before release 2.14. The default behaviour from release 2.14 onwards is to reject such input files, and so the --accept-unknown-input-arch option has been added to restore the old behaviour. |
| --as-needed  --no-as-needed | This option affects ELF DT\_NEEDED tags for dynamic libraries mentioned on the command line after the --as-needed option. Normally the linker will add a DT\_NEEDED tag for each dynamic library mentioned on the command line, regardless of whether the library is actually needed or not. --as-needed causes a DT\_NEEDED tag to only be emitted for a library that satisfies an undefined symbol reference from a regular object file or, if the library is not found in the DT\_NEEDED lists of other libraries linked up to that point, an undefined symbol reference from another dynamic library. --no-as-needed restores the default behaviour. |
| --add-needed  --no-add-needed | These two options have been deprecated because of the similarity of their names to the --as-needed and --no-as-needed options. They have been replaced by --copy-dt-needed-entries and --no-copy-dt-needed-entries. |
| -assert keyword | This option is ignored for SunOS compatibility. |
| -Bdynamic  -dy  -call\_shared | Link against dynamic libraries. This is only meaningful on platforms for which shared libraries are supported. This option is normally the default on such platforms. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for -l options which follow it. |
| -Bgroup | Set the "DF\_1\_GROUP" flag in the "DT\_FLAGS\_1" entry in the dynamic section. This causes the runtime linker to handle lookups in this object and its dependencies to be performed only inside the group. --unresolved-symbols=report-all is implied. This option is only meaningful on ELF platforms which support shared libraries. |
| -Bstatic  -dn  -non\_shared  -static | Do not link against shared libraries. This is only meaningful on platforms for which shared libraries are supported. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for -l options which follow it. This option also implies --unresolved-symbols=report-all. This option can be used with -shared. Doing so means that a shared library is being created but that all of the library's external references must be resolved by pulling in entries from static libraries. |
| -Bsymbolic | When creating a shared library, bind references to global symbols to the definition within the shared library, if any. Normally, it is possible for a program linked against a shared library to override the definition within the shared library. This option is only meaningful on ELF platforms which support shared libraries. |
| -Bsymbolic-functions | When creating a shared library, bind references to global function symbols to the definition within the shared library, if any. This option is only meaningful on ELF platforms which support shared libraries. |
| --dynamic-list=dynamic-list-file | Specify the name of a dynamic list file to the linker. This is typically used when creating shared libraries to specify a list of global symbols whose references shouldn't be bound to the definition within the shared library, or creating dynamically linked executables to specify a list of symbols which should be added to the symbol table in the executable. This option is only meaningful on ELF platforms which support shared libraries.  The format of the dynamic list is the same as the version node without scope and node name. |
| --dynamic-list-data | Include all global data symbols to the dynamic list. |
| --dynamic-list-cpp-new | Provide the builtin dynamic list for C++ operator new and delete. It is mainly useful for building shared libstdc++. |
| --dynamic-list-cpp-typeinfo | Provide the builtin dynamic list for C++ runtime type identification. |
| --check-sections  --no-check-sections | Asks the linker not to check section addresses after they have been assigned to see if there are any overlaps. Normally the linker will perform this check, and if it finds any overlaps it will produce suitable error messages. The linker does know about, and does make allowances for sections in overlays. The default behaviour can be restored by using the command line switch --check-sections. Section overlap is not usually checked for relocatable links. You can force checking in that case by using the --check-sections option. |
| --copy-dt-needed-entries  --no-copy-dt-needed-entries | This option affects the treatment of dynamic libraries referred to by DT\_NEEDED tags inside ELF dynamic libraries mentioned on the command line. Normally the linker won't add a DT\_NEEDED tag to the output binary for each library mentioned in a DT\_NEEDED tag in an input dynamic library. With --copy-dt-needed-entries specified on the command line, however, any dynamic libraries that follow it will have their DT\_NEEDED entries added. The default behaviour can be restored with --no-copy-dt-needed-entries.  This option also has an effect on the resolution of symbols in dynamic libraries. With --copy-dt-needed-entries dynamic libraries mentioned on the command line will be recursively searched, following their DT\_NEEDED tags to other libraries, to resolve symbols required by the output binary. With the default setting, however, the searching of dynamic libraries that follow it will stop with the dynamic library itself. No DT\_NEEDED links will be traversed to resolve symbols. |
| --cref | Output a cross reference table. If a linker map file is being generated, the cross reference table is printed to the map file. Otherwise, it is printed on the standard output.  The format of the table is intentionally simple, so that it may be easily processed by a script if necessary. The symbols are printed out, sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol. |
| --no-define-common | This option inhibits the assignment of addresses to common symbols. The script command "INHIBIT\_COMMON\_ALLOCATION" has the same effect.  The --no-define-common option allows decoupling the decision to assign addresses to Common symbols from the choice of the output file type; otherwise a non-Relocatable output type forces assigning addresses to Common symbols. Using --no-define-common allows Common symbols that are referenced from a shared library to be assigned addresses only in the main program. This eliminates the unused duplicate space in the shared library, and also prevents any possible confusion over resolving to the wrong duplicate when there are many dynamic modules with specialized search paths for runtime symbol resolution. |
| --defsym=symbol=expression | Create a global symbol in the output file, containing the absolute address given by expression. You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the expression in this context: you may give a [hexadecimal](https://www.computerhope.com/jargon/h/hex.htm) constant or the name of an existing symbol, or use "+" and "-" to add or subtract hexadecimal constants or symbols. If you need more elaborate expressions, consider using the linker command language from a script. Note: there should be no white space between symbol, the equals sign ("="), and expression. |
| --demangle[=style]  --no-demangle | These options control whether to "demangle" symbol names in error messages and other output. When the linker is told to demangle, it tries to present symbol names in a readable fashion: it strips leading underscores if they are used by the object file format, and converts C++ mangled symbol names into user readable names. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler. The linker will demangle by default unless the environment variable COLLECT\_NO\_DEMANGLE is set. These options may be used to override the default. |
| -Ifile  --dynamic-linker=file | Set the name of the dynamic linker. This is only meaningful when generating dynamically linked ELF executables. The default dynamic linker is normally correct; don't use this unless you know what you are doing. |
| --fatal-warnings  --no-fatal-warnings | Treat all warnings as errors. The default behaviour can be restored with the option --no-fatal-warnings. |
| --force-exe-suffix | Make sure that an output file has a .exe suffix.  If a successfully built fully linked output file does not have a ".exe" or ".dll" suffix, this option forces the linker to copy the output file to one of the same name with a ".exe" suffix. This option is useful when using unmodified Unix makefiles on a [Microsoft Windows](https://www.computerhope.com/jargon/w/windows.htm) host, since some versions of Windows won't run an image unless it ends in a ".exe" suffix. |
| --gc-sections  --no-gc-sections | Enable garbage collection of unused input sections. It is ignored on targets that do not support this option. The default behaviour (of not performing this garbage collection) can be restored by specifying --no-gc-sections on the command line.  --gc-sections decides which input sections are used by examining symbols and relocations. The section containing the entry symbol and all sections containing symbols undefined on the command-line will be kept, as will sections containing symbols referenced by dynamic objects. Note that when building shared libraries, the linker must assume that any visible symbol is referenced. Once this initial set of sections has been determined, the linker recursively marks as used any section referenced by their relocations. See --entry and --undefined.  This option can be set when doing a partial link (enabled with option -r). In this case the root of symbols kept must be explicitly specified either by an --entry or --undefined option or by a "ENTRY" command in the linker script. |
| --print-gc-sections  --no-print-gc-sections | List all sections removed by garbage collection. The listing is printed on stderr. This option is only effective if garbage collection has been enabled via the --gc-sections) option. The default behaviour (of not listing the sections that are removed) can be restored by specifying --no-print-gc-sections on the command line. |
| --print-output-format | Print the name of the default output format (perhaps influenced by other command-line options). This is the string that would appear in an "OUTPUT\_FORMAT" linker script command. |
| --help | Print a summary of the command-line options on the standard output and exit. |
| --target-help | Print a summary of all target specific options on the standard output and exit. |
| -Map=mapfile | Print a link map to the file mapfile. See the description of the -Moption, above. |
| --no-keep-memory | ld normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This option tells ld to instead optimize for memory usage, by rereading the symbol tables as necessary. This may be required if ld runs out of memory space while linking a large executable. |
| --no-undefined  -z defs | Report unresolved symbol references from regular object files. This is done even if the linker is creating a non-symbolic shared library. The switch --[no-]allow-shlib-undefined controls the behaviour for reporting unresolved references found in shared libraries being linked in. |
| --allow-multiple-definition  -z muldefs | Normally when a symbol is defined multiple times, the linker will report a fatal error. These options allow multiple definitions and the first definition will be used. |
| --allow-shlib-undefined  --no-allow-shlib-undefined | Allows or disallows undefined symbols in shared libraries. This switch is similar to --no-undefined except that it determines the behaviour when the undefined symbols are in a shared library rather than a regular object file. It does not affect how undefined symbols in regular object files are handled.  The default behaviour is to report errors for any undefined symbols referenced in shared libraries if the linker is being used to create an executable, but to allow them if the linker is being used to create a shared library.  The reasons for allowing undefined symbol references in shared libraries specified at link time are that:   * A shared library specified at link time may not be the same as the one that is available at load time, so the symbol might actually be resolvable at load time. * There are some operating systems, e.g., [BeOS](https://www.computerhope.com/jargon/b/beos.htm)and HPPA, where undefined symbols in shared libraries are normal.   The BeOS kernel for example patches shared libraries at load time to select whichever function is most appropriate for the current architecture. This is used, for example, to dynamically select an appropriate memset function. |
| --no-undefined-version | Normally when a symbol has an undefined version, the linker will ignore it. This option disallows symbols with undefined version and a fatal error will be issued instead. |
| --default-symver | Create and use a default symbol version (the soname) for unversioned exported symbols. |
| --default-imported-symver | Create and use a default symbol version (the soname) for unversioned imported symbols. |
| --no-warn-mismatch | Normally ld will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endiannesses. This option tells ld that it should silently permit such possible errors. This option should only be used with care, in cases when you have taken some special action that ensures that the linker errors are inappropriate. |
| --no-warn-search-mismatch | Normally ld will give a warning if it finds an incompatible library during a library search. This option silences the warning. |
| --no-whole-archive | Turn off the effect of the --whole-archive option for subsequent archive files. |
| --noinhibit-exec | Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error whatsoever. |
| -nostdlib | Only search library directories explicitly specified on the command line. Library directories specified in linker scripts (including linker scripts specified on the command line) are ignored. |
| --oformat=output-format | ld may be configured to support more than one kind of object file. If your ld is configured this way, you can use the --oformat option to specify the binary format for the output object file. Even when ld is configured to support alternative object formats, you don't usually need to specify this, as ld should be configured to produce as a default output format the most usual format on each machine. output-format is a text string, the name of a particular format supported by the BFD libraries. You can list the available binary formats with objdump -i. The script command "OUTPUT\_FORMAT" can also specify the output format, but this option overrides it. |
| -pie  --pic-executable | Create a position-independent executable. This is currently only supported on ELF platforms. Position-independent executables are similar to shared libraries in that they are relocated by the dynamic linker to the virtual address the [OS](https://www.computerhope.com/os.htm) chooses for them (which can vary between invocations). Like normal dynamically linked executables they can be executed and symbols defined in the executable cannot be overridden by shared libraries. |
| -qmagic | This option is ignored for Linux compatibility. |
| -Qy | This option is ignored for SVR4 compatibility. |
| --relax  --no-relax | An option with machine dependent effects. This option is only supported on a few targets.  On some platforms the --relax option performs target specific, global optimizations that become possible when the linker resolves addressing in the program, such as relaxing address modes, synthesizing new instructions, selecting shorter version of current instructions, and combining constant values.  On some platforms these link time global optimizations may make symbolic debugging of the resulting executable impossible. This is known to be the case for the Matsushita MN10200 and MN10300 family of processors.  On platforms where this is not supported, --relax is accepted, but ignored.  On platforms where --relax is accepted the option --no-relax can be used to disable the feature. |
| --retain-symbols-file=filename | Retain only the symbols listed in the file filename, discarding all others. The filename is a flat file, with one symbol name per line. This option is especially useful in environments (such as VxWorks) where a large global symbol table is accumulated gradually, to conserve run-time memory.  --retain-symbols-file does not discard undefined symbols, or symbols needed for relocations.  You may only specify --retain-symbols-file once in the command line. It overrides -s and -S. |
| -rpath=dir | Add a directory to the runtime library search path. This is used when linking an ELF executable with shared objects. All -rpath arguments are concatenated and passed to the runtime linker, which uses them to locate shared objects at runtime. The -rpath option is also used when locating shared objects that are needed by shared objects explicitly included in the link; see the description of the -rpath-linkoption. If -rpath is not used when linking an ELF executable, the contents of the environment variable "LD\_RUN\_PATH" will be used if it is defined.  The -rpath option may also be used on SunOS. By default, on SunOS, the linker will form a runtime search patch out of all the -Loptions it is given. If a -rpath option is used, the runtime search path will be formed exclusively using the -rpath options, ignoring the -Loptions. This can be useful when using gcc, which adds many -Loptions which may be on NFS mounted file systems.  For compatibility with other ELF linkers, if the -R option is followed by a directory name, rather than a file name, it is treated as the -rpathoption. |
| -rpath-link=dir | When using ELF or SunOS, one shared library may require another. This happens when an "ld -shared" link includes a shared library as one of the input files.  When the linker encounters such a dependency when doing a non-shared, non-relocatable link, it will automatically try to locate the required shared library and include it in the link, if it is not included explicitly. In such a case, the -rpath-link option specifies the first set of directories to search. The -rpath-link option may specify a sequence of directory names either by specifying a list of names separated by colons, or by appearing multiple times.  This option should be used with caution as it overrides the search path that may have been hard compiled into a shared library. In such a case it is possible to use unintentionally a different search path than the runtime linker would do.  The linker uses the following search paths to locate required shared libraries:   1. Any directories specified by -rpath-linkoptions. 2. Any directories specified by -rpath options. The difference between -rpath and -rpath-link is that directories specified by -rpathoptions are included in the executable and used at runtime, whereas the -rpath-linkoption is only effective at link time. Searching -rpath in this way is only supported by native linkers and cross linkers which have been configured with the --with-sysroot option. 3. On an ELF system, for native linkers, if the -rpath and -rpath-link options were not used, search the contents of the environment variable "LD\_RUN\_PATH". 4. On SunOS, if the -rpath option was not used, search any directories specified using -Loptions. 5. For a native linker, the search the contents of the environment variable "LD\_LIBRARY\_PATH". 6. For a native ELF linker, the directories in "DT\_RUNPATH" or "DT\_RPATH" of a shared library are searched for shared libraries needed by it. The "DT\_RPATH" entries are ignored if "DT\_RUNPATH" entries exist. 7. The default directories, normally /lib and /usr/lib. 8. For a native linker on an ELF system, if the file /etc/ld.so.conf exists, the list of directories found in that file.   If the required shared library is not found, the linker will issue a warning and continue with the link. |
| -shared  -Bshareable | Create a shared library. This is currently only supported on ELF, XCOFF and SunOS platforms. On SunOS, the linker will automatically create a shared library if the -e option is not used and there are undefined symbols in the link. |
| --sort-common  --sort-common=ascending  --sort-common=descending | This option tells ld to sort the common symbols by alignment in ascending or descending order when it places them in the appropriate output sections. The symbol alignments considered are sixteen-byte or larger, eight-byte, four-byte, two-byte, and one-byte. This is to prevent gaps between symbols due to alignment constraints. If no sorting order is specified, then descending order is assumed. |
| --sort-section=name | This option will apply "SORT\_BY\_NAME" to all wildcard section patterns in the linker script. |
| --sort-section=alignment | This option will apply "SORT\_BY\_ALIGNMENT" to all wildcard section patterns in the linker script. |
| --split-by-file[=size] | Similar to --split-by-reloc but creates a new output section for each input file when size is reached. The size defaults to a size of 1 if not given. |
| --split-by-reloc[=count] | Tries to creates extra sections in the output file so that no single output section in the file contains more than count relocations. This is useful when generating huge relocatable files for downloading into certain real time kernels with the COFF object file format; since COFF cannot represent more than 65535 relocations in a single section. Note that this will fail to work with object file formats which do not support arbitrary sections. The linker will not split up individual input sections for redistribution, so if a single input section contains more than count relocations one output section will contain that many relocations. The count defaults to a value of 32768. |
| --stats | Compute and display statistics about the operation of the linker, such as execution time and memory usage. |
| --sysroot=directory | Use directory as the location of the sysroot, overriding the configure-time default. This option is only supported by linkers that were configured using --with-sysroot. |
| --traditional-format | For some targets, the output of ld is different in some ways from the output of some existing linker. This switch requests ld to use the traditional format instead.  For example, on SunOS, ld combines duplicate entries in the symbol string table. This can reduce the size of an output file with full debugging information by over 30 percent. Unfortunately, the SunOS dbx program can not read the resulting program (although gdb has no trouble). The --traditional-format switch tells ld to not combine duplicate entries. |
| --section-start=sectionname=org | Locate a section in the output file at the absolute address given by org. You may use this option as many times as necessary to locate multiple sections in the command line. The org must be a single hexadecimal [integer](https://www.computerhope.com/jargon/i/integer.htm); for compatibility with other linkers, you may omit the leading 0x usually associated with hexadecimal values. Note: there should be no white space between sectionname, the equals sign ("="), and org. |
| -Tbss=org  -Tdata=org  -Ttext=org | Same as --section-start, with ".bss", ".data" or ".text" as the sectionname. |
| -Ttext-segment=org | When creating an ELF executable or shared object, it will set the address of the first byte of the text segment. |
| -Trodata-segment=org | When creating an ELF executable or shared object for a target where the read-only data is in its own segment separate from the executable text, it will set the address of the first byte of the read-only data segment. |
| --unresolved-symbols=method | Determine how to handle unresolved symbols. There are four possible values for method:   |  |  | | --- | --- | | ignore-all | Do not report any unresolved symbols. | | report-all | Report all unresolved symbols. This is the default. | | ignore-in-object-files | Report unresolved symbols that are contained in shared libraries, but ignore them if they come from regular object files. | | ignore-in-shared-libs | Report unresolved symbols that come from regular object files, but ignore them if they come from shared libraries. This can be useful when creating a dynamic binary and it is known that all the shared libraries that it should be referencing are included on the linker's command line.  The behaviour for shared libraries on their own can also be controlled by the --[no-]allow-shlib-undefined option. |   Normally the linker will generate an error message for each reported unresolved symbol but the option --warn-unresolved-symbols can change this to a warning. |
| --dll-verbose  --verbose[=NUMBER] | Display the version number for ld and list the linker emulations supported. Display which input files can and cannot be opened. Display the linker script being used by the linker. If the optional NUMBER argument > 1, plugin symbol status will also be displayed. |
| --version-script=version-scriptfile | Specify the name of a version script to the linker. This is typically used when creating shared libraries to specify additional information about the version hierarchy for the library being created. This option is only fully supported on ELF platforms which support shared libraries. It is partially supported on PE platforms, which can use version scripts to filter symbol visibility in auto-export mode: any symbols marked local in the version script will not be exported. |
| --warn-common | Warn when a common symbol is combined with another common symbol or with a symbol definition. Unix linkers allow this somewhat sloppy practice, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practice, so you may get some warnings about symbols in the libraries as well as in your programs.  There are three kinds of global symbols, illustrated here by C examples:  int i = 1;  A definition, which goes in the initialized data section of the output file.  extern int i;  An undefined reference, which does not allocate space. There must be either a definition or a common symbol for the variable somewhere.  int i;  A common symbol. If there are only (one or more) common symbols for a variable, it goes in the uninitialized data area of the output file. The linker merges multiple common symbols for the same variable into a single symbol. If they are of different sizes, it picks the largest size. The linker turns a common symbol into a declaration, if there is a definition of the same variable.  The --warn-common option can produce five kinds of warnings. Each warning consists of a pair of lines: the first describes the symbol just encountered, and the second describes the previous symbol encountered with the same name. One or both of the two symbols will be a common symbol.  1. Turning a common symbol into a reference, because there is already a definition for the symbol.  <file>(<section>): warning: common of `<symbol>'  overridden by definition  <file>(<section>): warning: defined here  2. Turning a common symbol into a reference, because a later definition for the symbol is encountered. This is the same as the previous case, except that the symbols are encountered in a different order.  <file>(<section>): warning: definition of `<symbol>'  overriding common  <file>(<section>): warning: common is here  3. Merging a common symbol with a previous same-sized common symbol.  <file>(<section>): warning: multiple common  of `<symbol>'  <file>(<section>): warning: previous common is here  4. Merging a common symbol with a previous larger common symbol.  <file>(<section>): warning: common of `<symbol>'  overridden by larger common  <file>(<section>): warning: larger common is here  5. Merging a common symbol with a previous smaller common symbol. This is the same as the previous case, except that the symbols are encountered in a different order.  <file>(<section>): warning: common of `<symbol>'  overriding smaller common  <file>(<section>): warning: smaller common is here |
| --warn-constructors | Warn if any global constructors are used. This is only useful for a few object file formats. For formats like COFF or ELF, the linker can not detect the use of global constructors. |
| --warn-multiple-gp | Warn if multiple global pointer values are required in the output file. This is only meaningful for certain processors, such as the [DEC](https://www.computerhope.com/jargon/d/dec.htm) Alpha. Specifically, some processors put large-valued constants in a special section. A special [register](https://www.computerhope.com/jargon/r/register.htm) (the global pointer) points into the middle of this section, so that constants can be loaded efficiently via a base-register relative addressing mode. Since the offset in base-register relative mode is fixed and relatively small (e.g., 16 [bits](https://www.computerhope.com/jargon/b/bit.htm)), this limits the maximum size of the constant pool. Thus, in large programs, it is often necessary to use multiple global pointer values to be able to address all possible constants. This option causes a warning to be issued whenever this case occurs. |
| --warn-once | Only warn once for each undefined symbol, rather than once per module which refers to it. |
| --warn-section-align | Warn if the address of an output section is changed because of alignment. Typically, the alignment will be set by an input section. The address will only be changed if it not explicitly specified; that is, if the "SECTIONS" command does not specify a start address for the section. |
| --warn-shared-textrel | Warn if the linker adds a DT\_TEXTREL to a shared object. |
| --warn-alternate-em | Warn if an object has alternate ELF [machine code](https://www.computerhope.com/jargon/m/machlang.htm). |
| --warn-unresolved-symbols | If the linker is going to report an unresolved symbol (see the option --unresolved-symbols) it will normally generate an error. This option makes it generate a warning instead. |
| --error-unresolved-symbols | This restores the linker's default behaviour of generating errors when it is reporting unresolved symbols. |
| --whole-archive | For each archive mentioned on the command line after the --whole-archive option, include every object file in the archive in the link, rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared library, forcing every object to be included in the resulting shared library. This option may be used more than once.  Two notes when using this option from gcc: First, gcc doesn't know about this option, so you have to use -Wl,-whole-archive. Second, don't forget to use -Wl,-no-whole-archive after your list of archives, because gcc will add its own list of archives to your link and you may not want this flag to affect those as well. |
| --wrap=symbol | Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to "\_\_wrap\_symbol". Any undefined reference to "\_\_real\_symbol" will be resolved to symbol.  This can be used to provide a wrapper for a system function. The wrapper function should be called "\_\_wrap\_symbol". If it wishes to call the system function, it should call "\_\_real\_symbol".  Here is a trivial example:  void \*  \_\_wrap\_malloc (size\_t c)  {  printf ("malloc called with %zu\n", c);  return \_\_real\_malloc (c);  }  If you link other code with this file using --wrap malloc, then all calls to "malloc" will call the function "\_\_wrap\_malloc" instead. The call to "\_\_real\_malloc" in "\_\_wrap\_malloc" will call the real "malloc" function.  You may want to provide a "\_\_real\_malloc" function as well, so that links without the --wrap option will succeed. If you do this, you should not put the definition of "\_\_real\_malloc" in the same file as "\_\_wrap\_malloc"; if you do, the assembler may resolve the call before the linker has a chance to wrap it to "malloc". |
| --eh-frame-hdr | Request creation of ".eh\_frame\_hdr" section and ELF "PT\_GNU\_EH\_FRAME" segment header. |
| --no-ld-generated-unwind-info | Request creation of ".eh\_frame" unwind info for linker-generated code sections like PLT. This option is on by default if linker-generated unwind info is supported. |
| --enable-new-dtags  --disable-new-dtags | This linker can create the new dynamic tags in ELF. But the older ELF systems may not understand them. If you specify --enable-new-dtags, the dynamic tags will be created as needed. If you specify --disable-new-dtags, no new dynamic tags will be created. By default, the new dynamic tags are not created. Note that those options are only available for ELF systems. |
| --hash-size=number | Set the default size of the linker's [hash](https://www.computerhope.com/jargon/h/hash.htm) tables to a [prime number](https://www.computerhope.com/jargon/p/prime-number.htm)close to number. Increasing this value can reduce the length of time it takes the linker to perform its tasks, at the expense of increasing the linker's memory requirements. Similarly reducing this value can reduce the memory requirements at the expense of speed. |
| --hash-style=style | Set the type of linker's hash table(s). style can be either "sysv" for classic ELF ".hash" section, "gnu" for new style GNU ".gnu.hash" section or "both" for both the classic ELF ".hash" and new style GNU ".gnu.hash" hash tables. The default is "sysv". |
| --reduce-memory-overheads | This option reduces memory requirements at ld runtime, at the expense of linking speed. This was introduced to select the old O(n^2) algorithm for link map file generation, rather than the new O(n) algorithm which uses about 40% more memory for symbol storage.  Another effect of the switch is to set the default hash table size to 1021, which again saves memory at the cost of lengthening the linker's run time. This is not done, however, if the --hash-size switch has been used.  The --reduce-memory-overheads switch may be also be used to enable other tradeoffs in future versions of the linker. |
| --build-id  --build-id=style | Request creation of ".note.gnu.build-id" ELF note section. The contents of the note are unique bits identifying this linked file. The style can be "uuid" to use 128 random bits, "sha1" to use a 160-bit SHA1 [hash](https://www.computerhope.com/jargon/h/hashing.htm) on the normative parts of the output contents, "md5" to use a 128-bit MD5 hash on the normative parts of the output contents, or "0xhexstring" to use a chosen bit string specified as an even number of hexadecimal digits ("-" and ":" characters between digit pairs are ignored). If style is omitted, "sha1" is used.  The "md5" and "sha1" styles produces an identifier that is always the same in an identical output file, but will be unique among all nonidentical output files. It is not intended to be compared as a [checksum](https://www.computerhope.com/jargon/c/checksum.htm) for the file's contents. A linked file may be changed later by other tools, but the build ID bit string identifying the original linked file does not change.  Passing "none" for style disables the setting from any "--build-id" options earlier on the command line. |

The i386 PE linker supports the -shared option, which causes the output to be a dynamically linked library ([DLL](https://www.computerhope.com/jargon/d/dll.htm)) instead of a normal executable. You should name the output "\*.dll" when you use this option. Also, the linker fully supports the standard "\*.def" files, which may be specified on the linker command line like an object file (in fact, it should precede archives it exports symbols from, to ensure that they get linked in, just like a normal object file).  
  
In addition to the options common to all targets, the i386 PE linker support additional command line options that are specific to the i386 PE target. Options that take values may be separated from their values by either a space or an equals sign.

|  |  |
| --- | --- |
| --add-stdcall-alias | If given, symbols with a stdcall suffix (@nn) will be exported as-is and also with the suffix stripped. [This option is specific to the i386 PE targeted port of the linker] |
| --base-file file | Use file as the name of a file in which to save the base addresses of all the relocations needed for generating DLLs with dlltool. [This is an i386 PE specific option] |
| --dll | Create a DLL instead of a regular executable. You may also use -shared or specify a "LIBRARY" in a given ".def" file. [This option is specific to the i386 PE targeted port of the linker] |
| --enable-long-section-names  --disable-long-section-names | The PE variants of the Coff object format add an extension that permits the use of section names longer than eight characters, the normal limit for Coff. By default, these names are only allowed in object files, as fully-linked executable images do not carry the Coff string table required to support the longer names. As a GNU extension, it is possible to allow their use in executable images as well, or to (probably pointlessly) disallow it in object files, by using these two options. Executable images generated with these long section names are slightly non-standard, carrying as they do a string table, and may generate confusing output when examined with non-GNU PE-aware tools, such as file viewers and dumpers. However, GDB relies on the use of PE long section names to find Dwarf-2 debug information sections in an executable image at runtime, and so if neither option is specified on the command-line, ld will enable long section names, overriding the default and technically correct behaviour, when it finds the presence of debug information while linking an executable image and not stripping symbols. [This option is valid for all PE targeted ports of the linker] |
| --enable-stdcall-fixup  --disable-stdcall-fixup | If the link finds a symbol that it cannot resolve, it will attempt to do "fuzzy linking" by looking for another defined symbol that differs only in the format of the symbol name (cdecl vs stdcall) and will resolve that symbol by linking to the match. For example, the undefined symbol "\_foo" might be linked to the function "\_foo@12", or the undefined symbol "\_bar@16" might be linked to the function "\_bar". When the linker does this, it prints a warning, since it normally should have failed to link, but sometimes import libraries generated from third-party dlls may need this feature to be usable. If you specify --enable-stdcall-fixup, this feature is fully enabled and warnings are not printed. If you specify --disable-stdcall-fixup, this feature is disabled and such mismatches are considered to be errors. [This option is specific to the i386 PE targeted port of the linker] |
| --leading-underscore  --no-leading-underscore | For most targets default symbol-prefix is an underscore and is defined in target's description. By this option it is possible to disable/enable the default underscore symbol-prefix. |
| --export-all-symbols | If given, all global symbols in the objects used to build a DLL will be exported by the DLL. Note that this is the default if there otherwise wouldn't be any exported symbols. When symbols are explicitly exported via DEF files or implicitly exported via function attributes, the default is to not export anything else unless this option is given. Note that the symbols "DllMain@12", "DllEntryPoint@0", "DllMainCRTStartup@12", and "impure\_ptr" will not be automatically exported. Also, symbols imported from other DLLs will not be re-exported, nor will symbols specifying the DLL's internal layout such as those beginning with "\_head\_" or ending with "\_iname". Also, no symbols from "libgcc", "libstd++", "libmingw32", or "crtX.o" will be exported. Symbols whose names begin with "\_\_rtti\_" or "\_\_builtin\_" will not be exported, to help with C++ DLLs. Finally, there is an extensive list of cygwin-private symbols that are not exported (obviously, this applies on when building DLLs for cygwintargets). These cygwin-excludes are: "\_cygwin\_dll\_entry@12", "\_cygwin\_crt0\_common@8", "\_cygwin\_noncygwin\_dll\_entry@12", "\_fmode", "\_impure\_ptr", "cygwin\_attach\_dll", "cygwin\_premain0", "cygwin\_premain1", "cygwin\_premain2", "cygwin\_premain3", and "environ". [This option is specific to the i386 PE targeted port of the linker] |
| --exclude-symbolssymbol,symbol,... | Specifies a list of symbols which should not be automatically exported. The symbol names may be delimited by commas or colons. [This option is specific to the i386 PE targeted port of the linker] |
| --exclude-all-symbols | Specifies no symbols should be automatically exported. [This option is specific to the i386 PE targeted port of the linker] |
| --file-alignment | Specify the file alignment. Sections in the file will always begin at file offsets that are multiples of this number. This defaults to 512. [This option is specific to the i386 PE targeted port of the linker] |
| --heap reserve  --heapreserve,commit | Specify the number of bytes of memory to reserve (and optionally commit) to be used as heap for this program. The default is 1Mb reserved, 4K committed. [This option is specific to the i386 PE targeted port of the linker] |
| --image-basevalue | Use value as the base address of your program or dll. This is the lowest memory location that will be used when your program or dll is loaded. To reduce the need to relocate and improve performance of your dlls, each should have a unique base address and not overlap any other dlls. The default is 0x400000 for executables, and 0x10000000 for dlls. [This option is specific to the i386 PE targeted port of the linker] |
| --kill-at | If given, the stdcall suffixes (@nn) will be stripped from symbols before they are exported. [This option is specific to the i386 PE targeted port of the linker] |
| --large-address-aware | If given, the appropriate bit in the "Characteristics" field of the COFF header is set to indicate that this executable supports virtual addresses greater than 2 gigabytes. This should be used in conjunction with the /3GB or /USERVA=value megabytes switch in the "[operating systems]" section of the BOOT.INI. Otherwise, this bit has no effect. [This option is specific to PE targeted ports of the linker] |
| --major-image-version value | Sets the major number of the "image version". Defaults to 1. [This option is specific to the i386 PE targeted port of the linker] |
| --major-os-version value | Sets the major number of the "os version". Defaults to 4. [This option is specific to the i386 PE targeted port of the linker] |
| --major-subsystem-version value | Sets the major number of the "subsystem version". Defaults to 4. [This option is specific to the i386 PE targeted port of the linker] |
| --minor-image-version value | Sets the minor number of the "image version". Defaults to 0. [This option is specific to the i386 PE targeted port of the linker] |
| --minor-os-version value | Sets the minor number of the "os version". Defaults to 0. [This option is specific to the i386 PE targeted port of the linker] |
| --minor-subsystem-version value | Sets the minor number of the "subsystem version". Defaults to 0. [This option is specific to the i386 PE targeted port of the linker] |
| --output-def file | The linker will create the file file which will contain a DEF file corresponding to the DLL the linker is generating. This DEF file (which should be called "\*.def") may be used to create an import library with "dlltool" or may be used as a reference to automatically or implicitly exported symbols. [This option is specific to the i386 PE targeted port of the linker] |
| --out-implib file | The linker will create the file file which will contain an import lib corresponding to the DLL the linker is generating. This import lib (which should be called "\*.dll.a" or "\*.a" may be used to link clients against the generated DLL; this behaviour makes it possible to skip a separate "dlltool" import library creation step. [This option is specific to the i386 PE targeted port of the linker] |
| --enable-auto-image-base | Automatically choose the image base for DLLs, unless one is specified using the "--image-base" argument. By using a hash generated from the dllname to create unique image bases for each DLL, in-memory collisions and relocations which can delay program execution are avoided. [This option is specific to the i386 PE targeted port of the linker] |
| --disable-auto-image-base | Do not automatically generate a unique image base. If there is no user-specified image base ("--image-base") then use the platform default. [This option is specific to the i386 PE targeted port of the linker] |
| --dll-search-prefix string | When linking dynamically to a dll without an import library, search for "<string><basename>.dll" in preference to "lib<basename>.dll". This behaviour allows easy distinction between DLLs built for the various "subplatforms": native, cygwin, uwin, pw, etc. For instance, cygwin DLLs typically use "--dll-search-prefix=cyg". [This option is specific to the i386 PE targeted port of the linker] |
| --enable-auto-import | Do sophisticated linking of "\_symbol" to "\_\_imp\_\_symbol" for DATA imports from DLLs, and create the necessary thunking symbols when building the import libraries with those DATA exports. Note: Use of the 'auto-import' extension will cause the text section of the image file to be made writable. This does not conform to the PE-COFF format specification published by Microsoft.  Note: use of the 'auto-import' extension will also cause read only data which would normally be placed into the .rdata section to be placed into the .datasection instead. This is to work around a problem with consts that is described [at the cygwin website](https://www.cygwin.com/ml/cygwin/2004-09/msg01101.html).  Using 'auto-import' generally will 'just work', but sometimes you may see this message:  "variable '<var>' can't be auto-imported.  Please read the documentation for  ld's "--enable-auto-import" for details."  This message occurs when some (sub)expression accesses an address ultimately given by the sum of two constants (Win32 import tables only allow one). Instances where this may occur include accesses to member fields of struct variables imported from a DLL, as well as using a constant index into an array variable imported from a DLL. Any multiword variable (arrays, structs, long long, etc) may trigger this error condition. However, regardless of the exact data type of the offending exported variable, ld will always detect it, issue the warning, and exit.  There are several ways to address this difficulty, regardless of the data type of the exported variable:  One way is to use --enable-runtime-pseudo-reloc switch. This leaves the task of adjusting references in your client code for runtime environment, so this method works only when runtime environment supports this feature.  A second solution is to force one of the 'constants' to be a variable -- that is, unknown and un-optimizable at compile time. For arrays, there are two possibilities: a) make the indexee (the array's address) a variable, or b) make the 'constant' index a variable. Thus:  extern type extern\_array[];  extern\_array[1] -->  { volatile type \*t=extern\_array; t[1] }  or  extern type extern\_array[];  extern\_array[1] -->  { volatile int t=1; extern\_array[t] }  For structs (and most other multiword data types) the only option is to make the struct itself (or the long long, etc.) variable:  extern struct s extern\_struct;  extern\_struct.field -->  { volatile struct s \*t=&extern\_struct; t->field }  or  extern long long extern\_ll;  extern\_ll -->  { volatile long long \* local\_ll=&extern\_ll; \*local\_ll }  A third method of dealing with this difficulty is to abandon 'auto-import' for the offending symbol and mark it with "\_\_declspec(dllimport)". However, in practice that requires using compile-time #defines to indicate whether you are building a DLL, building client code that will link to the DLL, or merely building/linking to a static library. In making the choice between the various methods of resolving the 'direct address with constant offset' problem, you should consider typical real-world usage:  Original:  foo.h:  extern int arr[];  foo.c:  #include "foo.h"  void main(int argc, char \*\*argv){  printf("%d\n",arr[1]);  }  Solution 1:  foo.h:  extern int arr[];  foo.c:  #include "foo.h"  void main(int argc, char \*\*argv){  printf("%d\n",arr[1]);  }  Solution 2:  foo.h:  extern int arr[];  foo.c:  #include "foo.h"  void main(int argc, char \*\*argv){  /\* This workaround is for win32  and cygwin; do not "optimize" \*/  volatile int \*parr = arr;  printf("%d\n",parr[1]);  }  A fourth way to avoid this problem is to re-code your library to use a functional interface rather than a data interface for the offending variables (e.g., set\_foo() and get\_foo() accessory functions). [This option is specific to the i386 PE targeted port of the linker] |
| --disable-auto-import | Do not attempt to do sophisticated linking of "\_symbol" to "\_\_imp\_\_symbol" for DATA imports from DLLs. [This option is specific to the i386 PE targeted port of the linker] |
| --enable-runtime-pseudo-reloc | If your code contains expressions described in --enable-auto-import section, that is, DATA imports from DLL with non-zero offset, this switch will create a vector of 'runtime pseudo relocations' which can be used by runtime environment to adjust references to such data in your client code. [This option is specific to the i386 PE targeted port of the linker] |
| --disable-runtime-pseudo-reloc | Do not create pseudo relocations for non-zero offset DATA imports from DLLs. This is the default. [This option is specific to the i386 PE targeted port of the linker] |
| --enable-extra-pe-debug | Show additional debug info related to auto-import symbol thunking. [This option is specific to the i386 PE targeted port of the linker] |
| --section-alignment | Sets the section alignment. Sections in memory will always begin at addresses that are a multiple of this number. Defaults to 0x1000. [This option is specific to the i386 PE targeted port of the linker] |
| --stack reserve  --stackreserve,commit | Specify the number of bytes of memory to reserve (and optionally commit) to be used as stack for this program. The default is 2Mb reserved, 4K committed. [This option is specific to the i386 PE targeted port of the linker] |
| --subsystemwhich  --subsystemwhich:major  --subsystemwhich:major.minor | Specifies the subsystem under which your program will execute. The legal values for which are "native", "windows", "console", "posix", and "xbox". You may optionally set the subsystem version also. Numeric values are also accepted for which. [This option is specific to the i386 PE targeted port of the linker]  The following options set flags in the "DllCharacteristics" field of the PE file header: [These options are specific to PE targeted ports of the linker] |
| --dynamicbase | The image base address may be relocated using address space layout randomization (ASLR). This feature was introduced with MS Windows Vista for i386 PE targets. |
| --forceinteg | Code integrity checks are enforced. |
| --nxcompat | The image is compatible with the Data Execution Prevention. This feature was introduced with MS Windows XP SP2 for i386 PE targets. |
| --no-isolation | Although the image understands isolation, do not isolate the image. |
| --no-seh | The image does not use SEH. No SE handler may be called from this image. |
| --no-bind | Do not bind this image. |
| --wdmdriver | The driver uses the MS Windows Driver Model. |
| --tsaware | The image is Terminal Server aware. |

The C6X uClinux target uses a binary format called DSBT to support shared libraries. Each shared library in the system needs to have a unique index; all executables use an index of 0.

|  |  |
| --- | --- |
| --dsbt-sizesize | This option sets the number of entires in the DSBT of the current executable or shared library to size. The default is to create a table with 64 entries. |
| --dsbt-indexindex | This option sets the DSBT index of the current executable or shared library to index. The default is 0, which is appropriate for generating executables. If a shared library is generated with a DSBT index of 0, the "R\_C6000\_DSBT\_INDEX" relocs are copied into the output file.  The --no-merge-exidx-entries switch disables the merging of adjacent exidx entries in frame unwind info. |

The 68HC11 and 68HC12 linkers support specific options to control the memory bank switching mapping and trampoline code generation.

|  |  |
| --- | --- |
| --no-trampoline | This option disables the generation of trampoline. By default, a trampoline is generated for each far function that is called using a "jsr" instruction (this happens when a pointer to a far function is taken). |
| --bank-windowname | This option indicates to the linker the name of the memory region in the MEMORY specification that describes the memory bank window. The definition of such region is then used by the linker to compute paging and addresses within the memory window. |

RESULT:

Thus ld commands are studied and implemented in Unix/Linux