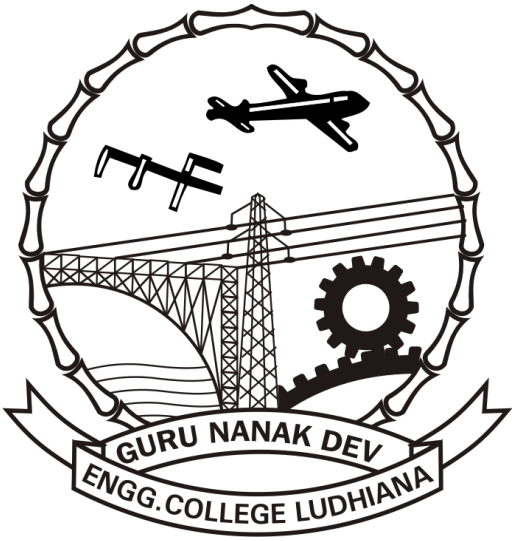
INSTRUCTION MANUAL

SYSTEM PROGRAMMING LAB

(BTCS-409)



Prepared by

Er. Blossom
Assistant Professor (CSE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GURU NANAK DEV ENGINEERING COLLEGE
LUDHIANA – 141006

DECLARATION

This Manual of System Programming Lab (BTCS-409) has been prepared by me as per syllabus of System Programming Lab (BTCS-409).

Signature

INDEX

Practical # 1

1. Create a menu driven interface for

a) Displaying contents of a file page wise.

b) Counting vowels, characters, and lines in a file.

c) Copying a file.

```
#include<fstream.h>
#include<conio.h>
#include<process.h>
void main()
{
 char q;
 z:
 clrscr();
 cout<<"Enter 1 to display the contents of a file"<<endl;
 cout<<"Enter 2 to count the number of characters, lines and vowels"<<endl;
 cout<<"Enter 3 to copy the contents of a file"<<endl;
 cout<<"Enter 4 to exit"<<endl;
 cin>>q;
 clrscr();
 switch(q)
 {
  case '1' :          //Used to display the contents of file//
  {
   int count=0,n=50;
   char source[40];
   cout<<"Enter the file name to display its contents"<<endl;
   cin>>source;
   ifstream MyFile(source);
   char ch;
   while(!MyFile.eof())
   {
     MyFile.get (ch);
     cout<<ch;
     if(ch=='\n')
     {
       count++;
     }
     if(count==n)
     {
       getch();
       n=n+49;
     }
```

```
 }
 MyFile.close();
 getch();
 goto z;
}
case '2' :            //Count the number of characters, lines & vowels in file//
{
  int countn=0,count=0,vowels=0;
  char source[40];
  cout<<"Enter the file name to count the no. of characters , lines and vowels "<<endl;
  cin>>source;
  ifstream MyFile(source);
  char ch;
  while(!MyFile.eof())
  {
   MyFile.get (ch);
   if(ch=='\n')
   {
     countn++;
   }
   else
   {
    count++;
   }
   if(ch=='a'||ch=='e'||ch=='i'||ch=='o'||ch=='u'||ch=='A'||ch=='E'||ch=='I'||ch=='U'||ch=='O')
   {
     vowels++;
   }
 }
 cout<<"\nThe Number of character are : "<<count-1<<endl;
 cout<<"The Number of lines are  : "<<countn<<endl;
 cout<<"The number of vowels are : "<<vowels<<endl;
 MyFile.close();
 getch();
 goto z;
}
case '3' :              //Copying the file//
{
 char source[40],target[40];
 cout<<"Enter the source file name "<<endl;
 cin>>source;
 cout<<"enter the file name in which you want to copy the data "<<endl;
 cin>>target;
 ifstream stream1(source);
 ofstream stream2(target);
 stream2 <<stream1.rdbuf();
```

```
   cout<<"the file has been copied";
   getch();
   goto z;
  }
  case '4' :
  {
   exit (1);
  }
  default:
  {
   cout<<"Wrong number entered "<<endl;
   cout<<"Please try again"<<endl;
   getch();
   goto z;
  }
 }
}
```

OUTPUT:
Menu:

Enter 1 to display the cotents of a file
Enter 2 to count the number of characters, lines and vowels
Enter 3 to copy the contnts of a file
Enter 4 to exit

Display:

Enter the file name to display its contents
ab.txt

Counting vowels, characters & Lines:

Enter the file name to count the no. of characters, lines and vowels
ab.txt

The Number of characters are : 3057
The Number of lines are : 60
The number of vowels are : 914

Copying a file:

Enter the source file name
ab.txt
enter the file name in which you want to copy the data

rb.txt
the file has been copied

Practical # 2

2. Write a program to check balance parenthesis of a given program. Also generate the error report.

```
#include<fstream.h>
#include<conio.h>
void main()
{
  clrscr();
  int count1=0,count2=0;
  char source[40];
  cout<<"Enter the file name to check balance in parentheses"<<endl;
  cin>>source;
  ifstream MyFile(source);
  char ch;
  while(!MyFile.eof())
  {
   MyFile.get (ch);

   if(ch=='{'||ch=='['||ch=='(')
   {
    count1++;
   }

   if(ch=='}'||ch==']'||ch==')')
   {
    count2++;
   }
  }

 if(count1==count2)
  {
   cout<<endl<<"There is balance in parentheses"<<endl;
  }
  else
   cout<<endl<<"There is imbalance in parentheses";
  MyFile.close();
  getch();
}
```

OUTPUT:

Enter the file name to check balance in parantheses
2nd.cpp
There is balance in parantheses

Practical # 3

3. To write a C program to generate the symbol table for the given assembly language.

Algorithm:

1. Start processing.

2. Declare structure for input and output files

3. Declare File pointers for input and output files

4. Open Input File(s) in Read mode and Open Output File(s) in write Mode.

5. Read the Intermediate File until EOF occurs.
5.1 If Symbol is not equal to NULL then

5.2 Write the Symbol Name and its address into Symbol table.
6. Close all the Files.

7. Print Symbol Table is created.

8. Stop processing.

Practical # 4

4. Write a program to create symbol table.

Algorithm:

1. open the file (assembly language program.)
2. Separate the mnemonic instructions into label , opcode and operand
3. Check whether the label is not empty
4. if true check whether opcode is START and store the label in symbol table and assign the operand as program counter value.
5. If opcode is RESB then store the label and program counter in label & value field of symbol table correspondingly. Add the operand with program counter.
6. If opcode is RESW then store the label and program counter in label & value field of symbol table correspondingly. Multiply the operand by 3 and Add the operand with program counter.
7. If opcode is WORD then store the label and program counter in label & value field of symbol table correspondingly. Increment the program counter by 3.
8. If opcode is BYTE then store the label and program counter in label & value field of symbol table correspondingly. Find the length of the constant and add it to program counter.
9. If opcede is EQU then store the label and program counter in label & value field of symbol table correspondingly.
10. if steps 4 to 9 fails then store the label and program counter in label & value field of symbol table correspondingly. . Increment the program counter by 3.
11. If the label is empty , Increment the program counter by 3.
12. Steps 2 to 10 are executed until EOF is encountered.
13. Display the content of symbol table.

Practical #5

5. To write a C program to Implement Single pass Assembler.

Algorithm:

1. Check whether the opcode is equal to START then assign operand address to pc and initialize the header record. Read the next line and initialize the new text record.
2. If the label is present in the read line then check the symbol table.
3. If found check whether the value is * then replace it by the current pc value.
4. Generate the new text record and store the forward list address and the current pc value in the record.
5. Otherwise insert the symbol and corresponding value in the symbol table.
6. Check whether the operand is present in the symbol table.
7. If present display its corresponding value in your text record.
8. If the symbol is not present insert the symbol in your symbol table and assign * for the address & increment the pc value by 1and insert this address in the forward list of symbol table.
9. Read the next line and repeat steps 2 to 8 until END is encountered.
10. If opcode is END then generate End record and assign the stating address.

Practical # 6

6. Study of LEX Tool

Theory:

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to LEX so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to LEX has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)*
This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:
• repetition, expressed by the "*****" operator
• alternation, expressed by the "|" operator
• concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state, and one or more final or accepting states.

start: goto state0
state0: read c
if c = letter goto state1
goto state0
state1: read c
if c = letter goto state1
if c = digit goto state1
goto state2
state2:accept string

This is the technique used by Lex. Regular expressions are translated by Lex to a computer program that mimics an FSA. Using the next *input* character, and *current state*, the next state is easily determined by indexing into a computer-generated state table.
Now we can easily understand some of Lex's limitations. For example, Lex cannot be used to

recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "**(**", we push it on the stack. When a "**)**" is encountered, we match it with the top of the stack, and pop the stack. However Lex only has states and transitions between states. Since it has no stack, it is not well suited for parsing nested structures. Yacc augments an FSA with a stack, and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

Table 1: Pattern Matching Primitives

Metacharacter	**Matches**
.	any character except newline
\n	newline
*****	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	**a** or **b**
(ab)+	one or more copies of **ab** (grouping)
"a+b"	literal "**a+b**" (C escapes still work)
[]	character class

Table 2: Pattern Matching Examples

Expression	**Matches**
abc	**abc**
abc*	**ab abc abcc abccc ...**
abc+	**abc abcc abccc ...**
a(bc)+	**abc abcbc abcbcbc ...**
a(bc)?	**a abc**
[abc]	one of: **a**, **b**, **c**
[a-z]	any letter, a-z
[a\-z]	one of: **a**, **-**, **z**

[-az]	one of: **-**, **a**, **z**
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: **a**, **b**
[a^b]	one of: **a**, ^, **b**
[a|b]	one of: **a**, |, **b**
a|b	one of: **a**, **b**

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class, normal operators lose their meaning.
Two operators allowed in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters, the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

... definitions ...
%%
... rules ...
%%
... subroutines ...

Input to Lex is divided into three sections, with **%%** dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

% %

Input is copied to output, one character at a time. The first **%%** is always required, as there must always be a rules section. However, if we don't specify any rules, then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example, with defaults explicitly coded:

%%

```
/* match everything except newline */
. ECHO;
/* match newline */
\n ECHO;
%%
int yywrap(void) {
return 1;
}
int main(void) {
yylex();
return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one.

This is followed by whitespace (space, tab or newline), and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "**.**" and "**\n**", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

#define ECHO fwrite(yytext, yyleng, 1, yyout)

Variable **yytext** is a pointer to the matched string (NULL-terminated), and **yyleng** is the length of the matched string. Variable **yyout** is the output file, and defaults to stdout. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done, or 0 if more processing is required. Every C program requires a **main** function. In this case, we simply call **yylex**, the main entry-point for lex. Some implementations of lex include copies of **main** and **yywrap** in a library, eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Table 3: Lex Predefined Variables

Name	**Function**
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	Length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	Output file
FILE *yyin	input file
INITIAL	Initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

Here is a program that does nothing at all. All input is matched, but no action is associated with any pattern, so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin**, and defaults to **stdin**.

```
%{
int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file, and must be bracketed with "**%{**" and "**%}**" markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit [0-9]
letter [A-Z, a-z]
%{
int count;
%}
%%
/* match identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
yylex();
printf("number of identifiers = %d\n", count);
return 0;

}
```

Practical# 7

7. Study of YACC Tool

Theory:

Grammars for yacc are described using a variant of Backus Naur Form (BNF). A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

E -> E + E
E -> E * E
E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as **E** (expression) are nonterminals. Terms such as **id** (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

E -> E * E (r2)
-> E * z (r3)
-> E + E * z (r1)
-> E + y * z (r3)
-> x + y * z (r3)

At each step we expanded a term, replacing the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression, we actually need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar, we need to reduce an expression to a single nonterminal. This is known as *bottom-up* or *shift-reduce* parsing, and uses a stack for storing terms. Here is the same derivation, but in reverse order:

1 . x + y * z shift
2 x . + y * z reduce(r3)
3 E . + y * z shift
4 E + . y * z shift
5 E + y . * z reduce(r3)
6 E + E . * z shift
7 E + E * . z shift
8 E + E * z . reduce(r3)
9 E + E * E . reduce(r2) emit multiply
10 E + E . reduce(r1) emit add
11 E . accept

Terms to the left of the dot are on the stack, while remaining input is to the right of the dot. We

start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production, we replace the matched tokens on the stack with the lhs of the production. Conceptually, the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a *handle*, and we are *reducing* the handle to the lhs of the production. This process continues until we have shifted all input to the stack, and only the starting nonterminal remains on the stack. In step 1 we shift the **x** to the stack. Step 2 applies rule r3 to the stack, changing **x** to **E**. We continue shifting and reducing, until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly, the add instruction is emitted in step 10. Thus, multiply has a higher precedence than addition.

Consider, however, the shift at step 6. Instead of shifting, we could have reduced, applying rule r1. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict. Our grammar is *ambiguous*, as there is more than one possible derivation that will yield the expression. In this case, operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar, or supply yacc with directives that indicate which operator has precedence. The latter method is simpler, and will be demonstrated in the practice section.

The following grammar has a *reduce-reduce* conflict. With an **id** on the stack, we may reduce to **T**, or reduce to **E**.

E -> T
E -> id
T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts, yacc will shift. For reduce-reduce conflicts, it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

Practice, Part I

... definitions ...
%%
... rules ...
%%
... subroutines ...

Input to yacc is divided into three sections. The definitions section consists of token declarations, and C code bracketed by "**%{**" and "**%}**". The BNF grammar is placed in the rules section, and user subroutines are added in the subroutines section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

%token INTEGER

This definition declares an **INTEGER** token. When we run yacc, it generates a parser in file **y.tab.c**, and also creates an include file, **y.tab.h**:

#ifndef YYSTYPE
#define YYSTYPE int

```
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens, yacc calls **yylex**. Function **yylex** has a return type of int, and returns the token. Values associated with the token are returned by lex in variable **yylval**.
For example,

```
[0-9]+ {
yylval = atoi(yytext);
return INTEGER;
}
```

would store the value of the integer in **yylval**, and return token **INTEGER** to yacc. The type of **yylval** is determined by **YYSTYPE**. Since the default type is integer, this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+] return *yytext; /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258, as lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
[0-9]+ {
yylval = atoi(yytext);
return INTEGER;
}
[-+\n] return *yytext;
[ \t] ; /* skip whitespace */
. yyerror("invalid character");
%%
int yywrap(void) {
return 1;
}
```

Internally, yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals, and represents the current parsing state. The value stack is an array of **YYSTYPE** elements, and associates a value with each element in the parse stack. For example, when lex returns an **INTEGER** token, yacc shifts this token to the parse stack. At the same time, the corresponding **yylval** is shifted to the value stack. The parse and value stacks are always synchronized, so finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

```
%{
    int yylex(void);
    void yyerror(char *);
    %}
    %token INTEGER
    %%
    program:
    program expr '\n' { printf("%d\n", $2); }
    |
    ;
    expr:
    INTEGER { $$ = $1; }
    | expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    ;
    %%
    void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
    }
    int main(void) {
    yyparse();
    return 0;
    }
```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified, followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.
By utilizing left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected, we print the value of the expression. When we apply the rule

```
expr: expr '+' expr { $$ = $1 + $3; }
```

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case, we pop "**expr '+' expr**" and push "**expr**". We have reduced the stack by popping three terms off the stack, and pushing back one term. We may reference positions in the value stack in our C code by specifying "**$1**" for the first term on the right-hand side of the production, "**$2**" for the second, and so on. "**$$**" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. Thus, the parse and value stacks remain synchronized.

Practical # 8

8. Exploring various features of debug command.

Setting Breakpoints

To control where your program stops running you will have to set breakpoints. The simplest way to set a breakpoint is with *Debug-Toggle-Breakpoint* or F5 key. Move the cursor to line 28, scanf("%d %d", &A, &B);, and press F5. Turbo debugger highlights the line, indicating there is a breakpoint set on it. Now select Debug-Run or press Ctrl-F9 to execute your program without interruption. The program stops at line 28. What has happened is, the debugger has executed the intervening statements, from printf("Enter two integer numbers, A & B: "); to scanf("%d %d",&A, &B); . Press F5 again to remove the breakpoint.

Running and stopping the program

To run the program, choose *Debug-Step over* or press F8. This allows execution one line at a time, until the first breakpoint, in the program . This brings up a window, titled "TDDEMO1.EXE". The window displays the line *Enter two integer numbers, A & B*. This window is the program input/output window, through which you interact with the program.

The scanf("%d %d",&A,&B) statement is now high-lighted. Pressing F8, brings the program input/output screen to the foreground, with the cursor blinking after the words *Enter two integer numbers, A & B*. That is because the scanf statement is waiting for you to enter two numbers. Type two integers and press ENTER. This will cause the next line to be highlighted. This process of executing one program statement at a time is called Single stepping.

Tracing in procedures and functions

When debugging a program, it sometimes becomes necessary to check the inner workings of a function to ensure its correctness. Select Debug-Trace Into or press F7 when the program has stopped at a statement containing the function call. TDW traces into the function, if it is accessible to the debugger and highlights the first statement in the function. Set a breakpoint at the line sum = Add(A, B); In our program, the execution will now stop at this statement when resumed. This statement contains a call to the function Add. Press F7 to trace into function Add. The first statement return A + B; in the function is now highlighted. Press F8 to resume execution. The function now returns control to the calling routine. The statement diff = Subtract(A,B); , following the statement sum = Add(A,B); is now highlighted.

Examining program variables

The Watches window can be opened by selecting the *View-Options* and it shows the value of variables you specify. For example, to watch the value of the variable sum, move the cursor to the variable name on line 30. Then choose *Debug-Add watch* or press Ctrl F5. The

variable sum now appears in the Watches window along with its type and value. As you execute the program, Turbo debugger updates this value to reflect the variable's current value.

Examining simple C data objects

Once you have stopped your program, there are a number of ways of looking at data using the Inspect command. This very powerful facility lets you examine data structures in the same way that you visualize them when you write a program. The *Debug-Inspect...* command lets you examine any variable you specify. Suppose you want to look at the value of the variable prod Move the cursor below the variable and choose *Debug-Inspect...* An Inspector window pops up. The first line tells you the variable name and type of data stored in it; the second line shows its value and the address in memory.

Examining compound C data objects

A compound data object such as an array or structure, contains multiple components. The variable pairs is a compound object. It is an array of records. To examine its content, invoke the Inspector window as described above. The Inspector window shows the values of the 20 elements of pairs. Use the arrow keys to scroll through the 20 elements. The title of the window shows the name of the data you are inspecting. A new Inspector window appears, showing the contents of that element in the array. The Inspector window shows the contents of a record of type integerpair. If one of these were in turn a compound data object, you could issue an Inspect command and dig down further into the data structure.

Changing C data values

So far, you have learned how to look at data in the program. Here you will learn to change the value of data items. Use the arrow keys to go to line 33 in the source file. Place the cursor at the variable A and choose*Debug-Evaluate/Modify*. The window now shows the value of variable A. you can enter any C expression that evaluates to a number. Type A+10 and press ENTER. The new value evaluated can be seen in the middle Result pane. Now, in the bottom pane with title New Value, enter a new value and press ENTER. Variable A takes the new value.

Reference

This section is a reference to using the different commands in TDW. The commands are categorized into two groups namely, "Controlling program execution" and "Examining and modifying data," according to their functionality.

Controlling program execution

When you debug a program in TDW, you execute portions of it and check at stopping points to see if it is behaving correctly. TDW provides you with a number of ways to control your program's execution.
You can

- stepping over source code
- tracing into code
- setting conditional/unconditional breakpoints

The Debug menu and the Speedup menu contain all the commands that are needed to control the execution of a program.

The Debug menu
The Debug menu has a number of options for executing different parts of your program. Since these options are used frequently, most are available on function keys. Following is the description of some of these options. The letters in the brackets stand for function keys with which the command can be invoked.

Run [Ctrl-F9] The Run menu Runs your program without stopping. Control returns to TDW when one of the following events occurs:

- Your program terminates.
- A breakpoint with a break action is encountered.
- You interrupt execution with Ctrl-Break.

Trace Into [F7] Executes a single source line. If the current line contains a function call, then the debugger traces into the function if it is accessible to the TDW.

Step Over [F8] Executes a single source line, skipping over any function calls. TDW treats any function call in a line as part of the line. You do not end up at the start of one of the functions or procedures. Instead, you end up at the next line in the current routine or at the previous routine that called the current one.

Toggle BreakPoint [F5] When the cursor is on a source line, then a breakpoint is set at that line when this option is chosen. If it is already a breakpoint then the breakpoint is removed.

Find execution point Finds the point in the program where execution is to resume. This can be of use when debugging a large program.

Add breakpoint New breakpoints can be added during debugging and this option brings up a window. Options to set a conditional breakpoint under a specific condition are available in this window.

Pause program The program can be paused at any point in the debugging process with this option.

Terminate program [Ctrl-F2] If you are debugging a program that requires as much memory as possible to complie, then choose this option after the debugging session.

Add watch [Ctrl-F5] You could add a new watch variable during the debugging session. The display type of the variable, the radix etc. can be selected.

Evaluate/Modify Any expression can be evaluated except those that

- contain symbols or macros defined with # define

- local or static variables not in the scope of the function being executed
- function calls

Inspect... As previously explained variables and structures (both simple and complex) can be inspected during debugging.

Speedup Menu
This is a small menu that comes up when the right mouse button is clicked or when Alt-F10 is pressed.

Browse symbol You can choose to browse any variable or structure in your program. If the structure has more than one elements then you could further inspect any of them.

View

Message window Any messages that may appear during linking and running of the program shows up in this window.

Globals This window shows all the global variables and functions that have been defined in the program.

Watch This window shows all the variables and structures that are being watched in the debugging session. The values of the variables during the different stages of execution are shown.

Call Stack This window shows the contents of this program stack. The different function calls in the program can be seen as being the stack contents when the TDDEMO1.C program is exected.

Register This window displays the contents of the data, pointer, index, segment, and instruction pointer registers, as well as the settings of the status word or flags.

Event log This window displays the results of breakpoints, Windows messages, and output messages generated while using the debugger.