Lab Practical

Assignment 1: Design a suitable data structures and implement Pass-I and Pass-II of two pass assembler of pseudo machine. The output of Pass-I should be input for pass -II.

File: two_pass_assembler.cpp

```
#include <bits/stdc++.h>
using namespace std;

struct Opcode {
    string mnemonic;
    string code;
    string type; // IS = Imperative, AD = Assembler Directive, DL = Declarative
};

struct Symbol {
    string name;
    int address;
};

map<string, Opcode> OPTAB;
map<string, int> REGTAB;
vector<Symbol> SYMTAB;

int searchSymbol(string sym) {
    for (int i = 0; i < SYMTAB.size(); i++)
        if (SYMTAB[i].name == sym) return i;
    return -1;
```

```
}

void pass1() {
    ifstream fin("input.asm");
    ofstream ic("intermediate.txt");
    ofstream sym("symtab.txt");

    string label, opcode, operand;
    int LC = 0;

    while (fin >> label >> opcode >> operand) {
        if (opcode == "START") {
            LC = stoi(operand);
            ic << "(AD,01) (C," << operand << ")\n";
            continue;
        }

        if (OPTAB.find(opcode) != OPTAB.end()) {
            if (label != "**") {
                int idx = searchSymbol(label);
                if (idx == -1)
                    SYMTAB.push_back({label, LC});
                else
                    SYMTAB[idx].address = LC;
            }

            Opcode op = OPTAB[opcode];
            if (op.type == "IS") {
                ic << LC << " (" << op.type << "," << op.code << ") ";
                if (REGTAB.find(operand) != REGTAB.end())
                    ic << "(" << REGTAB[operand] << ")";
                else {
```

```
            int idx = searchSymbol(operand);
            if (idx == -1) {
                SYMTAB.push_back({operand, -1});
                idx = SYMTAB.size() - 1;
            }
            ic << "(S," << idx + 1 << ")";
        }
        ic << "\n";
        LC++;
    }
    else if (op.type == "DL") {
        if (opcode == "DC") {
            ic << LC << " (" << op.type << "," << op.code << ") (C," << operand << ")\n";
            LC++;
        }
        else if (opcode == "DS") {
            ic << LC << " (" << op.type << "," << op.code << ") (C," << operand << ")\n";
            LC += stoi(operand);
        }
    }
  }
  else if (opcode == "END") {
    ic << "(AD,02)\n";
    break;
  }
}

for (int i = 0; i < SYMTAB.size(); i++)
  sym << i + 1 << "\t" << SYMTAB[i].name << "\t" << SYMTAB[i].address << "\n";

fin.close();
ic.close();
```

```
    sym.close();
}

void pass2() {
    ifstream ic("intermediate.txt");
    ifstream sym("symtab.txt");
    ofstream out("machine_code.txt");

    map<int, int> symAddr;
    int idx; string name; int addr;
    while (sym >> idx >> name >> addr)
        symAddr[idx] = addr;

    string line;
    while (getline(ic, line)) {
        if (line.find("IS") != string::npos) {
            stringstream ss(line);
            int LC;
            string is, reg, symb;
            ss >> LC >> is >> reg >> symb;
            int regno = 0, symindex = 0;

            if (reg.find("(") != string::npos)
                regno = reg[1] - '0';
            if (symb.find(",") != string::npos)
                symindex = symb[3] - '0';

            out << LC << "\t" << regno << "\t" << symAddr[symindex] << "\n";
        }
    }

    ic.close();
```

```
    sym.close();
    out.close();
}

int main() {
    OPTAB = {
        {"STOP", {"STOP", "00", "IS"}},
        {"ADD", {"ADD", "01", "IS"}},
        {"SUB", {"SUB", "02", "IS"}},
        {"MULT", {"MULT", "03", "IS"}},
        {"MOVER", {"MOVER", "04", "IS"}},
        {"MOVEM", {"MOVEM", "05", "IS"}},
        {"COMP", {"COMP", "06", "IS"}},
        {"BC", {"BC", "07", "IS"}},
        {"DIV", {"DIV", "08", "IS"}},
        {"READ", {"READ", "09", "IS"}},
        {"PRINT", {"PRINT", "10", "IS"}},
        {"START", {"START", "01", "AD"}},
        {"END", {"END", "02", "AD"}},
        {"DS", {"DS", "01", "DL"}},
        {"DC", {"DC", "02", "DL"}}
    };

    REGTAB = { {"AREG", 1}, {"BREG", 2}, {"CREG", 3}, {"DREG", 4} };

    pass1();
    pass2();

    cout << "Pass-I and Pass-II completed successfully.\n";
    cout << "Generated files:\n";
    cout << "1. intermediate.txt\n2. symtab.txt\n3. machine_code.txt\n";
    return 0;
```

}

input.asm

** START 100

** READ A

** READ B

** ADD A

** SUB B

** STOP **

A DS 1

B DC 5

** END **

Step 2: Compile and Run

g++ two_pass_assembler.cpp -o two_pass_assembler

./two_pass_assembler

Assignment 2:[Data Structure and Implement Pass-I and Pass-II of suitable two pass macroprocessor]

Step 1: Create the Source File

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_MNT 100
#define MAX_MDT 1000
#define MAX_LINE 512
#define MAX_PARAMS 20

typedef struct {
    char name[64];
    int mdt_index;
    int param_count;
    char param_names[MAX_PARAMS][64];
} MNTEntry;

char mdt[MAX_MDT][MAX_LINE];
int mdt_count = 0;
MNTEntry mnt[MAX_MNT];
int mnt_count = 0;

void trim(char *s) {
    int i, j;
    for (i = 0; s[i] && isspace((unsigned char)s[i]); i++);
    for (j = strlen(s) - 1; j >= 0 && isspace((unsigned char)s[j]); j--);
    s[j+1] = '\0';
    memmove(s, s + i, strlen(s + i) + 1);
}

int find_mnt(char *name) {
```

```
    for (int i = 0; i < mnt_count; i++)
        if (strcmp(mnt[i].name, name) == 0) return i;
    return -1;
}

void pass1(const char *infile, const char *intermediate, const char *mntfile, const char
*mdtfile) {
    FILE *fin = fopen(infile, "r");
    FILE *finter = fopen(intermediate, "w");
    FILE *fmnt = fopen(mntfile, "w");
    FILE *fmdt = fopen(mdtfile, "w");
    if (!fin || !finter || !fmnt || !fmdt) { printf("File error\n"); exit(1); }

    char line[MAX_LINE];
    while (fgets(line, sizeof(line), fin)) {
        char temp[MAX_LINE];
        strcpy(temp, line);
        trim(temp);
        if (strcmp(temp, "") == 0) { fputs(line, finter); continue; }
        if (strcmp(temp, "MACRO") == 0) {
            if (!fgets(line, sizeof(line), fin)) break;
            char header[MAX_LINE];
            strcpy(header, line);
            trim(header);
            char *tok = strtok(header, " \t");
            char macname[64];
            strcpy(macname, tok ? tok : "");
            char params_part[MAX_LINE] = "";
            tok = strtok(NULL, "\n");
            if (tok) { strcpy(params_part, tok); trim(params_part); }
            int param_count = 0;
            if (strlen(params_part) > 0) {
```

```
    char *p = params_part;
    char *q = strtok(p, ",");
    while (q && param_count < MAX_PARAMS) {
        trim(q);
        if (q[0] == '&') memmove(q, q+1, strlen(q));
        strcpy(mnt[mnt_count].param_names[param_count], q);
        param_count++;
        q = strtok(NULL, ",");
    }
}
strcpy(mnt[mnt_count].name, macname);
mnt[mnt_count].mdt_index = mdt_count;
mnt[mnt_count].param_count = param_count;
for (int k = 0; k < param_count; k++) { /* already copied */ }
mnt_count++;

while (fgets(line, sizeof(line), fin)) {
    char body[MAX_LINE];
    strcpy(body, line);
    trim(body);
    if (strcmp(body, "MEND") == 0) {
        strcpy(mdt[mdt_count++], "MEND");
        break;
    } else {
        char processed[MAX_LINE] = "";
        char token[MAX_LINE];
        int i = 0;
        int len = strlen(body);
        while (i < len) {
            if (body[i] == '&') {
                int j = ++i;
                char pname[64] = "";
```

```
                    int idx = 0;
                    while (j < len && (isalnum((unsigned char)body[j]) || body[j]=='_'))
pname[idx++] = body[j++], i++;
                    pname[idx] = '\0';
                    int found = -1;
                    for (int p = 0; p < param_count; p++) if (strcmp(pname,
mnt[mnt_count-1].param_names[p])==0) { found = p; break; }
                    if (found!=-1) {
                        char repl[16];
                        sprintf(repl, "%%P%d", found+1);
                        strcat(processed, repl);
                    } else {
                        strcat(processed, "&");
                        strcat(processed, pname);
                    }
                } else {
                    int j = i;
                    while (j < len && body[j] != '&') j++;
                    strncat(processed, body + i, j - i);
                    i = j;
                }
            }
            strcpy(mdt[mdt_count++], processed);
        }
    }
  } else {
    fputs(line, finter);
  }
}

for (int i = 0; i < mnt_count; i++) {
  fprintf(fmnt, "%s %d %d", mnt[i].name, mnt[i].mdt_index, mnt[i].param_count);
```

```
        for (int p = 0; p < mnt[i].param_count; p++) fprintf(fmnt, " %s", mnt[i].param_names[p]);
        fprintf(fmnt, "\n");
    }
    for (int i = 0; i < mdt_count; i++) fprintf(fmdt, "%d\t%s\n", i, mdt[i]);

    fclose(fin); fclose(finter); fclose(fmnt); fclose(fmdt);
}

void pass2(const char *intermediate, const char *expanded) {
    FILE *fin = fopen(intermediate, "r");
    FILE *fout = fopen(expanded, "w");
    if (!fin || !fout) { printf("File error pass2\n"); exit(1); }

    char line[MAX_LINE];
    while (fgets(line, sizeof(line), fin)) {
        char copy[MAX_LINE];
        strcpy(copy, line);
        trim(copy);
        if (strcmp(copy, "") == 0) { fputs(line, fout); continue; }
        char *tok = strtok(copy, " \t");
        if (!tok) { fputs(line, fout); continue; }
        int idx = find_mnt(tok);
        if (idx == -1) {
            fputs(line, fout);
        } else {
            char args_part[MAX_LINE] = "";
            char *p = strtok(NULL, "\n");
            if (p) { strcpy(args_part, p); trim(args_part); }
            char actuals[MAX_PARAMS][128];
            int ac = 0;
            if (strlen(args_part) > 0) {
                char *q = strtok(args_part, ",");
```

```
            while (q && ac < MAX_PARAMS) { trim(q); strcpy(actuals[ac++], q); q =
strtok(NULL, ","); }
        }
        int mdt_idx = mnt[idx].mdt_index;
        for (int k = mdt_idx; k < mdt_count; k++) {
            if (strcmp(mdt[k], "MEND") == 0) break;
            char outline[MAX_LINE] = "";
            char *s = mdt[k];
            int i = 0;
            while (s[i]) {
                if (s[i] == '%' && s[i+1] == 'P') {
                    i += 2;
                    int num = 0;
                    while (isdigit((unsigned char)s[i])) { num = num*10 + (s[i]-'0'); i++; }
                    if (num >=1 && num <= ac) strcat(outline, actuals[num-1]);
                } else {
                    int j = i;
                    while (s[j] && !(s[j]=='%' && s[j+1]=='P')) j++;
                    strncat(outline, s + i, j - i);
                    i = j;
                }
            }
            fprintf(fout, "%s\n", outline);
        }
    }
  }

  fclose(fin); fclose(fout);
}

int main(int argc, char *argv[]) {
  if (argc < 2) { printf("Usage: %s <inputfile>\n", argv[0]); return 1; }
```

```
    const char *infile = argv[1];
    pass1(infile, "intermediate.asm", "mnt.txt", "mdt.txt");
    pass2("intermediate.asm", "expanded.asm");
    printf("Pass-I done: mnt.txt, mdt.txt, intermediate.asm created\n");
    printf("Pass-II done: expanded.asm created\n");
    return 0;
}
```

Step 2: Create the Input Assembly File

In the same folder, create a new text file named input.asm

```
START 100
MACRO
INCR &X
LDA &X
ADD =1
STA &X
MEND
MACRO
SWAP &A,&B
LDA &A
STA TEMP
LDA &B
STA &A
LDA TEMP
```

```
STA &B
MEND
INCR A
INCR B
SWAP X,Y
END
```

Step 3: Compile the Program

```
gcc twopass_macro.c -o twopass_macro
```

```
./twopass_macro input.asm
```

Assignment 3: [Dynamic Link llbrary and Loading under linux platform]

Step 1: Create a C source file for library

File: mathlib.c

```
#include <stdio.h>
#include <math.h>

double add(double a, double b) {
    return a + b;
}
```

```
double subtract(double a, double b) {
    return a - b;
}

double multiply(double a, double b) {
    return a * b;
}

double divide(double a, double b) {
    if (b == 0) {
        printf("Error: Division by zero!\n");
        return 0;
    }
    return a / b;
}

double sine(double x) {
    return sin(x);
}

double cosine(double x) {
    return cos(x);
}
```

Step 2: Compile as a shared library (Linux)

```
gcc -fPIC -shared -o libmathlib.so mathlib.c -lm
```

Step 3: Create an application to test it

File: testapp.c

```
#include <stdio.h>
#include <dlfcn.h>

int main() {
    void *handle;
    double (*add)(double, double);
    double (*sine)(double);
    char *error;

    handle = dlopen("./libmathlib.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        return 1;
    }

    add = dlsym(handle, "add");
    sine = dlsym(handle, "sine");

    if ((error = dlerror()) != NULL)  {
        fprintf(stderr, "%s\n", error);
        return 1;
    }

    printf("5 + 3 = %.2f\n", (*add)(5, 3));
```

```
    printf("sin(90°) = %.2f\n", (*sine)(M_PI / 2));

    dlclose(handle);
    return 0;
}
```

Step 4: Compile and Run

```
gcc testapp.c -ldl -o testapp
./testapp
```

Assignment 4: Infix expression using LEX and YAAC

Assignment 4:

Step 1: Create a Lex file (expr.l)

```
%{
#include "y.tab.h"
%}

%%
[0-9]+          { return NUMBER; }
[a-zA-Z][a-zA-Z0-9]*  { return ID; }
"("             { return LPAREN; }
")"             { return RPAREN; }
"*"             { return MUL; }
"/"             { return DIV; }
"+"              { return ADD; }
"-"             { return SUB; }
```

```
[ \t\n]          ;   // ignore whitespaces
.                { return INVALID; }
%%

int yywrap() { return 1; }
```

Step 2: Create a YACC file (expr.y)

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}

%token NUMBER ID ADD SUB MUL DIV LPAREN RPAREN INVALID

%%
start: expr { printf("Valid Infix Expression\n"); return 0; }
     ;

expr: expr ADD term
    | expr SUB term
    | term
    ;

term: term MUL factor
    | term DIV factor
```

```
    | factor
    ;

factor: LPAREN expr RPAREN
      | NUMBER
      | ID
      ;
%%

void yyerror(const char *s) {
    printf("Invalid Infix Expression\n");
}

int main() {
    printf("Enter expression: ");
    yyparse();
    return 0;
}
```

Step 3: Compile the Program

```
yacc -d expr.y
lex expr.l
gcc lex.yy.c y.tab.c -o expr
```

Step 4: Run the Program

```
./expr
Enter expression: (a+b)*c
Valid Infix Expression
```

Step 5: Try Invalid Cases

Enter expression: a+*b

Invalid Infix Expression

Assignment 5 : Producer - Consumer Problem [Write a program to solve classical problem of snchronizaton using mute and semaphore]

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty;   // Counts empty slots
sem_t full;    // Counts filled slots
pthread_mutex_t mutex;

void* producer(void* arg) {
    int item = 1;
    while (1) {
        sem_wait(&empty);             // Wait if buffer is full
        pthread_mutex_lock(&mutex);   // Lock critical section

        buffer[in] = item;
```

```
        printf("Producer produced item %d at position %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;
        item++;

        pthread_mutex_unlock(&mutex); // Unlock
        sem_post(&full);              // Increase full slots
        sleep(1);
    }
}

void* consumer(void* arg) {
    while (1) {
        sem_wait(&full);              // Wait if buffer is empty
        pthread_mutex_lock(&mutex);   // Lock critical section

        int item = buffer[out];
        printf("Consumer consumed item %d from position %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex); // Unlock
        sem_post(&empty);             // Increase empty slots
        sleep(2);
    }
}

int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
```

```
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

Modified Version — Stops After N Items

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5
#define TOTAL_ITEMS 10

int buffer[BUFFER_SIZE];
int in = 0, out = 0;
int count = 0;
```

```
sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    int item = 1;
    while (item <= TOTAL_ITEMS) {
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);

        buffer[in] = item;
        printf("Producer produced item %d at position %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&full);
        item++;
        sleep(1);
    }
    return NULL;
}

void* consumer(void* arg) {
    int consumed = 0;
    while (consumed < TOTAL_ITEMS) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int item = buffer[out];
        printf("Consumer consumed item %d from position %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;
```

```
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        consumed++;
        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    printf("\nAll %d items produced and consumed successfully!\n", TOTAL_ITEMS);
    return 0;
}
```

User Input Driven

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <cstdlib> // Added for malloc() and free()

int *buffer;
int BUFFER_SIZE, TOTAL_ITEMS;
int in = 0, out = 0;

sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    int item = 1;
    while (item <= TOTAL_ITEMS) {
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);

        buffer[in] = item;
        printf("Producer produced item %d at position %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&full);
        item++;
        sleep(1);
    }
    return NULL;
}
```

```
void* consumer(void* arg) {
    int consumed = 0;
    while (consumed < TOTAL_ITEMS) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int item = buffer[out];
        printf("Consumer consumed item %d from position %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        consumed++;
        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    printf("Enter buffer size: ");
    scanf("%d", &BUFFER_SIZE);

    printf("Enter total items to produce/consume: ");
    scanf("%d", &TOTAL_ITEMS);

    buffer = (int*)malloc(BUFFER_SIZE * sizeof(int));

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
```

```
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    printf("\nAll %d items produced and consumed successfully!\n", TOTAL_ITEMS);
    free(buffer);
    return 0;
}
```

Sample Input

Enter buffer size: 5

Enter total items to produce/consume: 10

File Practical

1.Design suitable data structures and implement Pass-I of a two pass assembler for pseudo machine.

Implementation should consists of a few instructions from each category and a few assembler directives.

2.To design and implement Pass-II of a two Pass assembler for a Pseudo machine . Pass-II should utilise the intermediate code and symbol table generated by Pass -I to produce final machine code.

Program A — Pass-I (save as pass1_assembler.cpp)

```
// pass1_assembler.cpp
// Pass-I: builds OPTAB, SYMTAB, LITTAB and writes intermediate.txt, symtab.csv, littab.csv

#include <bits/stdc++.h>
using namespace std;

struct OptabEntry { string cls; int code; }; // cls: IS/DL/AD

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // ---------- Sample assembly program (editable) ----------
    vector<string> program = {
        "START 100",
        "FIRST   MOVER  AREG,='5'",
        "        ADD    BREG, ='1'",
        "        SUB    AREG, TWO",
        "        MULT   CREG,='2'",
        "        ORIGIN 200",
        "TWO     DC     2",
        "        DS     2",
        "        LTORG",
        "        MOVEM  DREG, TEMP",
```

```
    "TEMP    DS     1",
    "        END"
};

// ---------- OPTAB (instruction name -> (class, opcode)) ----------
unordered_map<string, OptabEntry> OPTAB = {
    {"STOP", {"IS",0}},
    {"MOVER", {"IS",1}},
    {"MOVEM", {"IS",2}},
    {"ADD", {"IS",3}},
    {"SUB", {"IS",4}},
    {"MULT", {"IS",5}},
    {"DIV", {"IS",6}},
    {"BC", {"IS",7}},
    {"COMP", {"IS",8}},
    {"READ", {"IS",9}},
    {"PRINT", {"IS",10}},
    // Declaratives
    {"DC", {"DL",1}},
    {"DS", {"DL",2}},
    // Assembler directives
    {"START", {"AD",1}},
    {"END", {"AD",2}},
    {"ORIGIN", {"AD",3}},
    {"EQU", {"AD",4}},
    {"LTORG", {"AD",5}}
};

unordered_map<string,int> REG = {{"AREG",1},{"BREG",2},{"CREG",3},{"DREG",4}};

// ---------- Data structures ----------
map<string,int> SYMTAB; // symbol -> address (if undefined yet use -1)
```

```
vector<pair<string,int>> SYM_INDEX; // index->(symbol,address)
vector<pair<string,int>> LITTAB; // (literal, addr) addr=-1 if not assigned
vector<string> INTERMEDIATE_LINES;

auto add_symbol_if_new = [&](const string &sym) {
    if (SYMTAB.find(sym) == SYMTAB.end()) {
        int idx = SYM_INDEX.size();
        SYMTAB[sym] = idx;
        SYM_INDEX.push_back({sym, -1});
    }
};

auto set_symbol_address = [&](const string &sym, int addr) {
    if (SYMTAB.find(sym) == SYMTAB.end()) {
        int idx = SYM_INDEX.size();
        SYMTAB[sym] = idx;
        SYM_INDEX.push_back({sym, addr});
        SYMTAB[sym] = idx;
    } else {
        int idx = SYMTAB[sym];
        SYM_INDEX[idx].second = addr;
    }
};

auto add_literal_if_new = [&](const string &lit) {
    for (int i=0;i<(int)LITTAB.size();++i) if (LITTAB[i].first==lit) return i;
    int idx = LITTAB.size();
    LITTAB.push_back({lit, -1});
    return idx;
};

// helper trim
```

```
auto trim = [](string s)->string{
    const char* ws = " \t\r\n";
    size_t b = s.find_first_not_of(ws);
    if (b==string::npos) return "";
    size_t e = s.find_last_not_of(ws);
    return s.substr(b,e-b+1);
};

int LC = 0;
bool start_seen = false;
int current_literal_pool_start = 0; // index in LITTAB
// Process source lines
for (int i=0;i<program.size();++i) {
    string line = trim(program[i]);
    if (line.empty()) continue;
    // tokenise by spaces, but keep commas and literals attached
    stringstream ss(line);
    vector<string> tokens;
    string tok;
    while (ss >> tok) tokens.push_back(tok);

    // detect label if first token not in OPTAB and not a directive/instruction
    string first_upper = tokens[0];
    for (auto &c:first_upper) c=toupper(c);
    bool hasLabel = false;
    string label;
    string opcode;
    vector<string> operands;

    // Determine if first token is opcode or label
    string t0 = tokens[0];
    string t0_up = t0; for (auto &c:t0_up) c = toupper(c);
```

```
    if (OPTAB.find(t0_up)==OPTAB.end() && !(t0_up=="START"||t0_up=="END"||
t0_up=="LTORG"||t0_up=="ORIGIN"||t0_up=="EQU")) {
        // treat as label
        hasLabel = true;
        label = tokens[0];
        // rest tokens
        if (tokens.size()>=2) {
            opcode = tokens[1];
            for (size_t k=2;k<tokens.size();++k) operands.push_back(tokens[k]);
        } else {
            // label without opcode - skip
            continue;
        }
    } else {
        opcode = tokens[0];
        for (size_t k=1;k<tokens.size();++k) operands.push_back(tokens[k]);
    }

    // normalize opcode uppercase
    string opcode_up = opcode; for (auto &c:opcode_up) c=toupper(c);

    // Handle label definition
    if (hasLabel) {
        // assign current LC to label
        set_symbol_address(label, LC);
    }

    // Handle directives and instructions
    if (opcode_up == "START") {
        if (!operands.empty()) {
            string addr = operands[0];
            LC = stoi(addr);
```

```
            start_seen = true;
        } else LC = 0;
        // write intermediate entry for START as AD
        string out = to_string(LC) + "\tAD,1";
        if (operands.size()) out += "\tC," + operands[0];
        INTERMEDIATE_LINES.push_back(out);
        continue;
    }

    if (opcode_up == "END" || opcode_up == "LTORG") {
        // allocate literals from current pool start to end
        for (int li = current_literal_pool_start; li < (int)LITTAB.size(); ++li) {
            if (LITTAB[li].second == -1) {
                LITTAB[li].second = LC;
                // write a literal as a declarative (DL,1 DC)
                string out = to_string(LC) + "\tDL,1\tC," + LITTAB[li].first.substr(1); // literal like
=5, remove '='
                INTERMEDIATE_LINES.push_back(out);
                LC += 1;
            }
        }
        current_literal_pool_start = LITTAB.size();
        // write AD entry
        string out = to_string(LC) + "\tAD," + (opcode_up=="END"? "2":"5");
        INTERMEDIATE_LINES.push_back(out);
        if (opcode_up=="END") break;
        continue;
    }

    if (opcode_up == "ORIGIN") {
        // operand may be expression or number (simple number supported)
        if (!operands.empty()) {
```

```
            string expr = operands[0];
            // for simplicity support numeric constants only and symbol +/- const
            int newlc = 0;
            if (isdigit(expr[0])) newlc = stoi(expr);
            else {
                // symbol +/- value
                size_t plus = expr.find('+');
                size_t minus = expr.find('-');
                if (plus!=string::npos) {
                    string sym = expr.substr(0,plus);
                    int off = stoi(expr.substr(plus+1));
                    if (SYMTAB.find(sym)!=SYMTAB.end()) {
                        int idx = SYMTAB[sym]; newlc = SYM_INDEX[idx].second + off;
                    } else newlc = off; // fallback
                } else if (minus!=string::npos) {
                    string sym = expr.substr(0,minus);
                    int off = stoi(expr.substr(minus+1));
                    if (SYMTAB.find(sym)!=SYMTAB.end()) {
                        int idx = SYMTAB[sym]; newlc = SYM_INDEX[idx].second - off;
                    } else newlc = -off;
                } else {
                    if (SYMTAB.find(expr)!=SYMTAB.end()) newlc =
SYM_INDEX[SYMTAB[expr]].second;
                }
            }
            LC = newlc;
        }
        string out = to_string(LC) + "\tAD,3";
        if (!operands.empty()) out += "\t" + string("C,") + operands[0];
        INTERMEDIATE_LINES.push_back(out);
        continue;
    }
```

```
        if (opcode_up == "EQU") {
            // label EQU expr
            if (hasLabel && !operands.empty()) {
                string expr=operands[0];
                int val=0;
                if (isdigit(expr[0])) val=stoi(expr);
                else {
                    // symbol
                    if (SYMTAB.find(expr)!=SYMTAB.end()) val =
SYM_INDEX[SYMTAB[expr]].second;
                    else val = 0;
                }
                set_symbol_address(label, val);
                string out = to_string(LC) + "\tAD,4\tC,"+to_string(val);
                INTERMEDIATE_LINES.push_back(out);
            }
            continue;
        }

        // Declarative DL: DC / DS
        if (OPTAB.find(opcode_up)!=OPTAB.end() && OPTAB[opcode_up].cls=="DL") {
            if (opcode_up=="DC") {
                // define constant - operand is value
                int val = 0;
                if (!operands.empty()) {
                    string op0 = operands[0];
                    // remove possible commas
                    if (op0.back()==',') op0.pop_back();
                    val = stoi(op0);
                }
                // if label present it already set above
```

```
        string out = to_string(LC) + "\tDL,1\tC," + to_string(val);
        INTERMEDIATE_LINES.push_back(out);
        LC += 1;
    } else if (opcode_up=="DS") {
        int size = 1;
        if (!operands.empty()) {
            string op0 = operands[0];
            if (op0.back()==',') op0.pop_back();
            size = stoi(op0);
        }
        string out = to_string(LC) + "\tDL,2\tC," + to_string(size);
        INTERMEDIATE_LINES.push_back(out);
        LC += size;
    }
    continue;
}

// Imperative statements
if (OPTAB.find(opcode_up)!=OPTAB.end() && OPTAB[opcode_up].cls=="IS") {
    int opcode_num = OPTAB[opcode_up].code;
    // parse operands (register, symbol/literal/constant)
    string op1type="", op1val="";
    string op2type="", op2val="";
    // join operands tokens back by spaces then split by comma
    string opsJoined;
    for (size_t k=0;k<operands.size();++k) {
        if (k) opsJoined += " ";
        opsJoined += operands[k];
    }
    // split by comma
    vector<string> ops;
    {
```

```
    string cur;
    for (char c: opsJoined) {
        if (c==',') { if (!cur.empty()) { ops.push_back(trim(cur)); cur.clear(); } }
        else cur.push_back(c);
    }
    if (!cur.empty()) ops.push_back(trim(cur));
}
if (ops.size()>=1) {
    string o = ops[0];
    // register?
    string upo=o; for (auto &c:upo) c=toupper(c);
    if (REG.find(upo)!=REG.end()) {
        op1type="RG"; op1val = to_string(REG[upo]);
    } else {
        // could be literal or symbol or constant
        if (!o.empty() && o[0]=='=') {
            int li = add_literal_if_new(o);
            op1type="L"; op1val = to_string(li);
        } else if (isdigit(o[0]) || (o.size()>1 && o[0]=='-' && isdigit(o[1]))) {
            op1type="C"; op1val = o;
        } else {
            add_symbol_if_new(o);
            op1type="S"; op1val = to_string(SYMTAB[o]);
        }
    }
}
if (ops.size()>=2) {
    string o = ops[1];
    string upo=o; for (auto &c:upo) c=toupper(c);
    if (REG.find(upo)!=REG.end()) {
        op2type="RG"; op2val = to_string(REG[upo]);
    } else {
```

```
                if (!o.empty() && o[0]=='=') {
                    int li = add_literal_if_new(o);
                    op2type="L"; op2val = to_string(li);
                } else if (isdigit(o[0]) || (o.size()>1 && o[0]=='-' && isdigit(o[1]))) {
                    op2type="C"; op2val = o;
                } else {
                    add_symbol_if_new(o);
                    op2type="S"; op2val = to_string(SYMTAB[o]);
                }
            }
        }
        // write intermediate: LC \t IS,code \t op1type,op1val \t op2type,op2val
        string out = to_string(LC) + "\tIS," + (opcode_num<10? "0"+to_string(opcode_num):
to_string(opcode_num));
        if (!op1type.empty()) out += "\t" + op1type + "," + op1val;
        if (!op2type.empty()) out += "\t" + op2type + "," + op2val;
        INTERMEDIATE_LINES.push_back(out);
        LC += 1;
        continue;
    }

    // Unknown token - copy as comment or ignore
    // For safety, write as comment AD
    string out = to_string(LC) + "\tAD,0\tC,0";
    INTERMEDIATE_LINES.push_back(out);
}

// At end if any literals left unallocated, allocate them
for (int li=current_literal_pool_start; li<(int)LITTAB.size(); ++li) {
    if (LITTAB[li].second==-1) {
        LITTAB[li].second = LC;
        string out = to_string(LC) + "\tDL,1\tC," + LITTAB[li].first.substr(1);
```

```
            INTERMEDIATE_LINES.push_back(out);
            LC += 1;
        }
    }

    // Write outputs to files
    {
        ofstream f("intermediate.txt");
        for (auto &ln: INTERMEDIATE_LINES) f << ln << "\n";
    }
    {
        ofstream f("symtab.csv");
        f << "Index,Symbol,Address\n";
        for (int i=0;i<SYM_INDEX.size();++i) {
            f << i << "," << SYM_INDEX[i].first << "," << SYM_INDEX[i].second << "\n";
        }
    }
    {
        ofstream f("littab.csv");
        f << "Index,Literal,Address\n";
        for (int i=0;i<LITTAB.size();++i) {
            f << i << "," << LITTAB[i].first << "," << LITTAB[i].second << "\n";
        }
    }

    // Print summary to console
    cout << "\n--- INTERMEDIATE ---\n";
    for (auto &ln: INTERMEDIATE_LINES) cout << ln << "\n";

    cout << "\n--- SYMTAB ---\n";
    for (int i=0;i<SYM_INDEX.size();++i) cout << i << "\t" << SYM_INDEX[i].first << "\t" <<
SYM_INDEX[i].second << "\n";
```

```
    cout << "\n--- LITTAB ---\n";
    for (int i=0;i<LITTAB.size();++i) cout << i << "\t" << LITTAB[i].first << "\t" << LITTAB[i].second << "\n";

    cout << "\nPass-I finished. Files written: intermediate.txt, symtab.csv, littab.csv\n";
    return 0;
}
```

```
g++ -std=c++11 pass1_assembler.cpp -o pass1_assembler
./pass1_assembler
```

Pass 2————————————————————

Program B — Pass-II (save as pass2_assembler.cpp)

```
// pass2_assembler.cpp
// Pass-II: reads intermediate.txt, symtab.csv, littab.csv and produces machine_code.txt
#include <bits/stdc++.h>
using namespace std;

struct OptabEntry { string cls; int code; };

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Recreate same OPTAB and REG mapping used in Pass-I
```

```
unordered_map<string, OptabEntry> OPTAB = {
    {"STOP", {"IS",0}},
    {"MOVER", {"IS",1}},
    {"MOVEM", {"IS",2}},
    {"ADD", {"IS",3}},
    {"SUB", {"IS",4}},
    {"MULT", {"IS",5}},
    {"DIV", {"IS",6}},
    {"BC", {"IS",7}},
    {"COMP", {"IS",8}},
    {"READ", {"IS",9}},
    {"PRINT", {"IS",10}},
    {"DC", {"DL",1}},
    {"DS", {"DL",2}},
    {"START", {"AD",1}},
    {"END", {"AD",2}},
    {"ORIGIN", {"AD",3}},
    {"EQU", {"AD",4}},
    {"LTORG", {"AD",5}}
};
unordered_map<string,int> REG = {{"AREG",1},{"BREG",2},{"CREG",3},{"DREG",4}};

// Read symtab.csv into vector index->(symbol,address)
vector<pair<string,int>> SYMTAB;
{
    ifstream f("symtab.csv");
    if (!f) { cerr << "symtab.csv not found. Run Pass-I first.\n"; return 1; }
    string line; getline(f,line); // header
    while (getline(f,line)) {
        if (line.empty()) continue;
        stringstream ss(line);
        string idx,sym,addr;
```

```
        getline(ss, idx, ',');
        getline(ss, sym, ',');
        getline(ss, addr, ',');
        int a = stoi(addr);
        SYMTAB.push_back({sym,a});
    }
}

// Read littab.csv
vector<pair<string,int>> LITTAB;
{
    ifstream f("littab.csv");
    if (!f) { cerr << "littab.csv not found. Run Pass-I first.\n"; return 1; }
    string line; getline(f,line);
    while (getline(f,line)) {
        if (line.empty()) continue;
        stringstream ss(line);
        string idx,lit,addr;
        getline(ss, idx, ',');
        getline(ss, lit, ',');
        getline(ss, addr, ',');
        int a = stoi(addr);
        LITTAB.push_back({lit,a});
    }
}

// Read intermediate.txt
vector<string> intermediate;
{
    ifstream f("intermediate.txt");
    if (!f) { cerr << "intermediate.txt not found. Run Pass-I first.\n"; return 1; }
    string line;
```

```
    while (getline(f,line)) {
        if (!line.empty()) intermediate.push_back(line);
    }
}

// Helper: trim
auto trim = [](string s)->string{
    const char* ws = " \t\r\n";
    size_t b = s.find_first_not_of(ws);
    if (b==string::npos) return "";
    size_t e = s.find_last_not_of(ws);
    return s.substr(b,e-b+1);
};

// Machine code output: each LC -> machine word(s)
vector<pair<int,string>> MACHINE; // (LC, code/text)

for (auto &ln : intermediate) {
    string line = trim(ln);
    if (line.empty()) continue;
    // tokens separated by tabs
    vector<string> parts;
    {
        string cur;
        stringstream ss(line);
        while (getline(ss, cur, '\t')) parts.push_back(trim(cur));
    }
    if (parts.size() < 1) continue;
    int LC = stoi(parts[0]);
    if (parts.size() >= 2) {
        string op = parts[1]; // like IS,01 or DL,1 or AD,1
        // parse op
```

```
size_t comma = op.find(',');
string cls = (comma==string::npos? op : op.substr(0,comma));
string code = (comma==string::npos? "" : op.substr(comma+1));
if (cls == "AD") {
    // assembler directive - no machine code
    continue;
} else if (cls == "DL") {
    // declarative
    // DL,1 -> DC value in parts[2] C,val
    if (code == "1") {
        // DC
        if (parts.size()>=3) {
            string vpart = parts[2];
            // expect C,val
            size_t cpos = vpart.find(',');
            if (cpos!=string::npos) {
                string type = vpart.substr(0,cpos);
                string val = vpart.substr(cpos+1);
                // write the constant as word
                string mc = "DATA " + val;
                MACHINE.push_back({LC, mc});
            }
        }
    } else if (code == "2") {
        // DS - reserve space => generate lines with 0 or keep as comment with size
        if (parts.size()>=3) {
            string vpart = parts[2];
            size_t cpos = vpart.find(',');
            if (cpos!=string::npos) {
                string val = vpart.substr(cpos+1);
                int sz = stoi(val);
                for (int k=0;k<sz;++k) MACHINE.push_back({LC+k, "RES 0"});
```

```
                }
            }
        }
    } else if (cls == "IS") {
        // example opcode code "01" numeric string
        int opc = stoi(code);
        // default register 0 and address 0
        int regnum = 0; int addrnum = 0;
        // parse operands if present
        if (parts.size()>=3) {
            string op1 = parts[2]; // like RG,1 or S,idx or L,idx or C,val
            size_t comma1 = op1.find(',');
            if (comma1!=string::npos) {
                string t1 = op1.substr(0,comma1);
                string v1 = op1.substr(comma1+1);
                if (t1=="RG") regnum = stoi(v1);
                else if (t1=="S") {
                    int symidx = stoi(v1);
                    if (symidx >= 0 && symidx < (int)SYMTAB.size()) addrnum =
SYMTAB[symidx].second;
                } else if (t1=="L") {
                    int litidx = stoi(v1);
                    if (litidx >=0 && litidx < (int)LITTAB.size()) addrnum =
LITTAB[litidx].second;
                } else if (t1=="C") {
                    addrnum = stoi(v1);
                }
            }
        }
        // sometimes second operand present (parts[3]) - rarely for our sample; handle if
memory operand in parts[3]
        if (parts.size()>=4) {
            string op2 = parts[3];
```

```
                size_t comma2 = op2.find(',');
                if (comma2!=string::npos) {
                    string t2 = op2.substr(0,comma2);
                    string v2 = op2.substr(comma2+1);
                    if (t2=="S") {
                        int symidx = stoi(v2);
                        if (symidx >= 0 && symidx < (int)SYMTAB.size()) addrnum =
SYMTAB[symidx].second;
                    } else if (t2=="L") {
                        int litidx = stoi(v2);
                        if (litidx >=0 && litidx < (int)LITTAB.size()) addrnum =
LITTAB[litidx].second;
                    } else if (t2=="C") {
                        addrnum = stoi(v2);
                    } else if (t2=="RG") {
                        regnum = stoi(v2);
                    }
                }
            }
            // Build machine word textually: OPC(2) RG(1) ADDR(3) - padded
            auto pad = [](int x, int w)->string{
                string s = to_string(x);
                while (s.size() < (size_t)w) s = "0"+s;
                return s;
            };
            string mc = pad(opc,2) + " " + to_string(regnum) + " " + pad(addrnum,3);
            MACHINE.push_back({LC, mc});
        }
    }
}

// Write machine_code.txt
```

```
    {
        ofstream out("machine_code.txt");
        out << "LC\tMACHINE_WORD\n";
        for (auto &p : MACHINE) out << p.first << "\t" << p.second << "\n";
    }

    // print to console
    cout << "\n--- MACHINE CODE ---\n";
    cout << "LC\tWORD\n";
    for (auto &p: MACHINE) cout << p.first << "\t" << p.second << "\n";

    cout << "\nPass-II complete. File written: machine_code.txt\n";
    return 0;
}
g++ -std=c++11 pass2_assembler.cpp -o pass2_assembler
./pass2_assembler
```

3.Design a suitable data structures and implement Pass-I of two Pass microprocessor . The output of Pass-I is MNT , MDT , and intermediate code file without any macro definitions.

4.To design a suitable data structures and implement Pass-II of two pass microprocessor use the MNT , MDT ,ALA build during Pass-I expand macro calls in the intermediate source program

3 & 4 combined

Save as macro_processor.cpp, compile and run.

```
// macro_processor.cpp
#include <bits/stdc++.h>
```

```
using namespace std;

/*
 Two-pass Macro Processor (Pass-I and Pass-II)
 Outputs: mnt.txt, mdt.txt, ala.txt, intermediate.txt, expanded.txt
*/

struct MNTEntry {
    string name;
    int mdtIndex;       // index in MDT where macro body starts
    int alaIndex;       // index in ALA table (we'll store ALA separately per macro)
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // ----------------------
    // SAMPLE SOURCE PROGRAM
    // ----------------------
    // You can edit this sample vector to test other inputs.
    vector<string> source = {
        "START 100",
        "MACRO",
        "INCR &ARG1,&ARG2",
        "    LDA &ARG1",
        "    ADD &ARG2",
        "    STA &ARG1",
        "MEND",
        "MACRO",
        "SWAP &X,&Y",
        "    TEMP DS 1",
```

```
        "    LDA &X",
        "    MOV R1, &X",
        "    MOV R2, &Y",
        "    MOV &X, R2",
        "    MOV &Y, R1",
        "MEND",
        "FIRST   INCR A, 1",
        "        SWAP B, C",
        "        INCR B, 2",
        "END"
    };

    // ----------------------
    // Data structures
    // ----------------------
    vector<MNTEntry> MNT;                   // Macro Name Table
    vector<string> MDT;                     // Macro Definition Table (stores parameterized lines)
    unordered_map<string, vector<string>> ALA; // ALA: macro name -> list of formal params (ordered)
    unordered_map<string, int> mntIndexLookup; // name -> index in MNT for quick lookup

    vector<string> intermediate; // intermediate source (macro definitions removed, CALL lines for invocations)

    // helpers
    auto trim = [](string s) {
        // trim both ends
        const char* ws = " \t\r\n";
        size_t b = s.find_first_not_of(ws);
        if (b==string::npos) return string();
        size_t e = s.find_last_not_of(ws);
        return s.substr(b, e-b+1);
```

```
};
auto split_comma = [&](const string &s)->vector<string>{
    vector<string> res;
    string cur;
    for (char c : s) {
        if (c==',') {
            res.push_back(trim(cur));
            cur.clear();
        } else cur.push_back(c);
    }
    if (!cur.empty()) res.push_back(trim(cur));
    return res;
};

// ----------------------
// PASS-I
// ----------------------
cout << "---- PASS-I: Building MNT, MDT, ALA, and Intermediate File ----\n";
int i = 0;
while (i < (int)source.size()) {
    string line = trim(source[i]);
    if (line.empty()) { ++i; continue; }

    string up = line;
    for (auto &c: up) c = toupper(c);

    if (up == "MACRO") {
        // Next line should be macro header: NAME &P1,&P2,...
        ++i;
        if (i >= (int)source.size()) {
            cerr << "Error: MACRO without header\n";
            return 1;
```

```
        }
        string header = trim(source[i]);
        // header: first token is macro name, remainder are params
        stringstream ss(header);
        string macname;
        ss >> macname;
        string rest;
        getline(ss, rest); // the rest contains formal parameters
        rest = trim(rest);

        vector<string> formals;
        if (!rest.empty()) {
            // remove possible commas at start
            if (rest.front()==',') rest = rest.substr(1);
            formals = split_comma(rest);
            // strip leading '&' if present
            for (auto &f : formals) {
                if (!f.empty() && f[0] == '&') f = f.substr(1);
            }
        }

        // Add MNT entry
        MNTEntry m;
        m.name = macname;
        m.mdtIndex = (int)MDT.size(); // macro body will start here
        m.alaIndex = (int)ALA.size(); // we will store ALA in ALA map keyed by name;
alaIndex is not strictly needed but kept for clarity
        MNT.push_back(m);
        mntIndexLookup[macname] = (int)MNT.size()-1;
        ALA[macname] = formals;
```

```
        // Read macro body until MEND, store in MDT replacing formals (&X) with positional markers #1 #2 ...
        ++i;
        while (i < (int)source.size()) {
            string bodyLine = source[i];
            string bodyTrim = trim(bodyLine);
            string upb = bodyTrim; for (auto &c: upb) c = toupper(c);
            if (upb == "MEND") {
                // store MEND marker in MDT (optionally)
                MDT.push_back("MEND");
                ++i;
                break;
            } else {
                // Replace occurrences of formal parameters (&NAME) in bodyLine with positional tokens #1, #2...
                string modLine = bodyLine;
                // For each formal, perform substitution for &formal
                for (int k = 0; k < (int)formals.size(); k++) {
                    string f = formals[k];
                    if (f.empty()) continue;
                    // replace occurrences of "&" + f with "#<k+1>"
                    string target = "&" + f;
                    string repl = "#" + to_string(k+1);
                    // naive replace all occurrences
                    size_t pos = 0;
                    while ((pos = modLine.find(target, pos)) != string::npos) {
                        modLine.replace(pos, target.length(), repl);
                        pos += repl.length();
                    }
                }
                MDT.push_back(modLine);
                ++i;
```

```
            }
        }
        // continue main loop without increment (we already advanced)
    } else {
        // Not a macro definition. Check if this line is a macro invocation.
        // Macro invocation in our simple syntax: first token is macro name and that name
exists in MNT.
        string firstTok;
        {
            stringstream ss(line);
            ss >> firstTok;
        }
        if (mntIndexLookup.find(firstTok) != mntIndexLookup.end()) {
            // It's a call. Extract actual parameters (rest of line)
            string rest;
            size_t pos = line.find(firstTok);
            if (pos != string::npos) {
                rest = line.substr(pos + firstTok.size());
            }
            rest = trim(rest);
            // parse actual params
            vector<string> actuals;
            if (!rest.empty()) {
                actuals = split_comma(rest);
            }
            // write a CALL entry into intermediate in a simple canonical form:
            // CALL <macname> , <arg1> , <arg2> ...
            string callLine = "CALL " + firstTok;
            if (!actuals.empty()) {
                callLine += " ";
                for (int k = 0; k < (int)actuals.size(); ++k) {
                    if (k) callLine += ",";
```

```
                callLine += actuals[k];
            }
        }
        intermediate.push_back(callLine);
    } else {
        // normal code line -> copy as-is
        intermediate.push_back(line);
    }
    ++i;
  }
}

// Save MNT, MDT, ALA, intermediate to files
{
  ofstream f("mnt.txt");
  f << "Index\tMacroName\tMDT_Index\n";
  for (int k = 0; k < (int)MNT.size(); ++k) {
    f << k << "\t" << MNT[k].name << "\t" << MNT[k].mdtIndex << "\n";
  }
  f.close();
}
{
  ofstream f("mdt.txt");
  f << "Index\tMDT_Line\n";
  for (int k = 0; k < (int)MDT.size(); ++k) {
    f << k << "\t" << MDT[k] << "\n";
  }
  f.close();
}
{
  ofstream f("ala.txt");
  f << "MacroName\tFormalParams(in-order)\n";
```

```
    for (auto &entry : ALA) {
        f << entry.first << "\t";
        for (int k = 0; k < (int)entry.second.size(); ++k) {
            if (k) f << ",";
            f << entry.second[k];
        }
        f << "\n";
    }
    f.close();
}
{
    ofstream f("intermediate.txt");
    for (auto &ln : intermediate) f << ln << "\n";
    f.close();
}

cout << "PASS-I complete. Files written: mnt.txt, mdt.txt, ala.txt, intermediate.txt\n";

// For demonstration print to console
cout << "\n--- MNT ---\n";
for (int k = 0; k < (int)MNT.size(); ++k) {
    cout << k << "\t" << MNT[k].name << "\tMDT idx: " << MNT[k].mdtIndex << "\n";
}
cout << "\n--- MDT ---\n";
for (int k = 0; k < (int)MDT.size(); ++k) {
    cout << k << "\t" << MDT[k] << "\n";
}
cout << "\n--- ALA ---\n";
for (auto &e : ALA) {
    cout << e.first << " : ";
    for (int k = 0; k < (int)e.second.size(); ++k) {
        if (k) cout << ",";
```

```
        cout << e.second[k];
    }
    cout << "\n";
}
cout << "\n--- INTERMEDIATE ---\n";
for (auto &ln : intermediate) cout << ln << "\n";

// ----------------------
// PASS-II (Expansion)
// ----------------------
cout << "\n---- PASS-II: Expanding Macros using MNT/MDT/ALA ----\n";
vector<string> expanded; // final expanded program

for (auto &line : intermediate) {
    string tline = trim(line);
    if (tline.empty()) continue;
    // detect CALL lines
    string up = tline;
    // uppercase check for 'CALL' token:
    {
        string tmp = up; for (auto &c: tmp) c = toupper(c);
        up = tmp;
    }
    if (up.rfind("CALL ", 0) == 0) {
        // parse: CALL NAME arg1,arg2,...
        string rest = trim(tline.substr(4)); // after CALL
        string macname;
        string argsPart;
        {
            stringstream ss(rest);
            ss >> macname;
            getline(ss, argsPart);
```

```
            argsPart = trim(argsPart);
        }
        vector<string> actuals;
        if (!argsPart.empty()) {
            actuals = split_comma(argsPart);
        }
        // locate macro in MNT
        if (mntIndexLookup.find(macname) == mntIndexLookup.end()) {
            cerr << "Error: CALL to unknown macro " << macname << "\n";
            continue;
        }
        int mntIdx = mntIndexLookup[macname];
        int mdtStart = MNT[mntIdx].mdtIndex;
        vector<string> formals = ALA[macname];

        // Build mapping #1 -> actuals[0], #2 -> actuals[1], etc.
        unordered_map<string,string> paramMap;
        for (int k = 0; k < (int)formals.size(); ++k) {
            string key = "#" + to_string(k+1);
            string val = (k < (int)actuals.size()) ? actuals[k] : ""; // if fewer actuals provided,
blank
            paramMap[key] = val;
        }

        // expand MDT lines starting from mdtStart until MEND
        for (int idx = mdtStart; idx < (int)MDT.size(); ++idx) {
            string mline = MDT[idx];
            if (trim(mline) == "MEND") break;
            // replace placeholders #i with actuals
            string outLine = mline;
            for (auto &p : paramMap) {
                // replace all occurrences of p.first with p.second
```

```
                size_t pos = 0;
                while ((pos = outLine.find(p.first, pos)) != string::npos) {
                    outLine.replace(pos, p.first.length(), p.second);
                    pos += p.second.length();
                }
            }
            // After param substitution, also tidy up leading/trailing spaces
            outLine = trim(outLine);
            expanded.push_back(outLine);
        }
    } else {
        // Normal line -> copy to expanded output
        expanded.push_back(line);
    }
}

// Save expanded output
{
    ofstream f("expanded.txt");
    for (auto &ln : expanded) f << ln << "\n";
    f.close();
}

cout << "PASS-II complete. File written: expanded.txt\n";
cout << "\n--- EXPANDED SOURCE ---\n";
for (auto &ln : expanded) cout << ln << "\n";

cout << "\nAll done.\n";
return 0;
}
```

Compile: g++ -std=c++11 macro_processor.cpp -o macro_processor

Run - ./macro_processor

3 & 4 Separately

```
#include <bits/stdc++.h>
using namespace std;

struct MNTEntry {
    string name;
    int mdtIndex;
};

int main() {
    vector<string> source = {
        "MACRO",
        "INCR &ARG1, &ARG2",
        "LOAD &ARG1",
        "ADD &ARG2",
        "STORE &ARG1",
        "MEND",
        "START",
        "CALL INCR A, B",
        "END"
    };

    vector<MNTEntry> MNT;
    vector<string> MDT;
    map<string, vector<string>> ALA;
```

```
vector<string> intermediate;

bool inMacro = false;
string currentMacro;

for (int i = 0; i < source.size(); i++) {
    string line = source[i];
    stringstream ss(line);
    string word;
    ss >> word;

    if (word == "MACRO") {
        inMacro = true;
        continue;
    }

    if (inMacro) {
        if (line == "MEND") {
            MDT.push_back("MEND");
            inMacro = false;
            continue;
        }

        // Macro prototype line
        if (currentMacro.empty()) {
            currentMacro = word;
            vector<string> params;
            string param;
            while (getline(ss, param, ',')) {
                param.erase(remove(param.begin(), param.end(), ' '), param.end());
                if (param[0] == '&') params.push_back(param);
            }
```

```
            ALA[currentMacro] = params;
            MNT.push_back({currentMacro, (int)MDT.size()});
        } else {
            // Inside macro body
            stringstream body(line);
            string token;
            string newLine;
            while (body >> token) {
                if (token[0] == '&') {
                    auto &params = ALA[currentMacro];
                    auto it = find(params.begin(), params.end(), token);
                    if (it != params.end()) {
                        int pos = distance(params.begin(), it);
                        token = "#" + to_string(pos + 1);
                    }
                }
                newLine += token + " ";
            }
            MDT.push_back(newLine);
        }
    } else {
        intermediate.push_back(line);
    }
}

// --- Output ---
cout << "\n===== PASS-I OUTPUT =====\n";
cout << "\nMNT (Macro Name Table):\n";
for (auto &e : MNT)
    cout << e.name << " -> MDT Index: " << e.mdtIndex << "\n";
```

```
    cout << "\nMDT (Macro Definition Table):\n";
    for (int i = 0; i < MDT.size(); i++)
        cout << i << ": " << MDT[i] << "\n";

    cout << "\nALA (Argument List Array):\n";
    for (auto &a : ALA) {
        cout << a.first << ": ";
        for (auto &p : a.second) cout << p << " ";
        cout << "\n";
    }

    cout << "\nIntermediate Code (without macro definitions):\n";
    for (auto &line : intermediate)
        cout << line << "\n";

    return 0;
}
```

4…….

```
#include <bits/stdc++.h>
using namespace std;

struct MNTEntry {
    string name;
    int mdtIndex;
};

int main() {
    // Data from Pass-I (normally read from files)
    vector<MNTEntry> MNT = {{"INCR", 0}};
```

```
    vector<string> MDT = {
        "LOAD #1",
        "ADD #2",
        "STORE #1",
        "MEND"
    };
    map<string, vector<string>> ALA = {
        {"INCR", {"&ARG1", "&ARG2"}}
    };

    vector<string> intermediate = {
        "START",
        "CALL INCR A, B",
        "END"
    };

    vector<string> expanded;

    for (auto &line : intermediate) {
        stringstream ss(line);
        string first;
        ss >> first;

        if (first == "CALL") {
            string macroName;
            ss >> macroName;
            auto it = find_if(MNT.begin(), MNT.end(), [&](MNTEntry e){return e.name ==
macroName;});
            if (it == MNT.end()) {
                expanded.push_back(line);
                continue;
            }
```

```
            vector<string> args;
            string arg;
            while (getline(ss, arg, ',')) {
                arg.erase(remove(arg.begin(), arg.end(), ' '), arg.end());
                if (!arg.empty()) args.push_back(arg);
            }

            int mdtIndex = it->mdtIndex;
            for (int i = mdtIndex; i < MDT.size() && MDT[i] != "MEND"; i++) {
                string expandedLine = MDT[i];
                for (int j = 0; j < args.size(); j++) {
                    string placeholder = "#" + to_string(j + 1);
                    size_t pos = expandedLine.find(placeholder);
                    if (pos != string::npos)
                        expandedLine.replace(pos, placeholder.size(), args[j]);
                }
                expanded.push_back(expandedLine);
            }
        } else {
            expanded.push_back(line);
        }
    }

    cout << "\n===== PASS-II OUTPUT =====\n";
    for (auto &line : expanded)
        cout << line << "\n";

    return 0;
}
```

```
g++ pass2_macro.cpp -o pass2_macro
./pass2_macro
```

5.The aim of this assignment is to study and understand CPU scheduling algorithm and their role in operating system process management . Specifically we will focus on two scheduling algorithm. First Come First Serve (FCFS) and Round Robin (RR) .The objective is to write , explain and analyse the algorithm with the help of practical example and comparison.

```
#include <bits/stdc++.h>
using namespace std;

struct Process {
    int pid;         // Process ID
    int burstTime;   // Burst Time
    int arrivalTime; // Arrival Time
    int waitingTime;
    int turnaroundTime;
    int remainingTime; // for RR
};

// Function to calculate FCFS Scheduling
void FCFS(vector<Process> processes) {
    int n = processes.size();
    int currentTime = 0;
    float totalWaiting = 0, totalTurnaround = 0;
```

```
    // Sort by arrival time
    sort(processes.begin(), processes.end(), [](Process a, Process b) {
        return a.arrivalTime < b.arrivalTime;
    });

    cout << "\n===== First Come First Serve (FCFS) Scheduling =====\n";
    cout << "Gantt Chart: ";

    for (int i = 0; i < n; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        cout << "| P" << processes[i].pid << " ";
        processes[i].waitingTime = currentTime - processes[i].arrivalTime;
        currentTime += processes[i].burstTime;
        processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;

        totalWaiting += processes[i].waitingTime;
        totalTurnaround += processes[i].turnaroundTime;
    }

    cout << "|\n";
    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
    cout << "-----------------------------------------\n";
    for (auto &p : processes) {
        cout << "P" << p.pid << "\t" << p.arrivalTime << "\t"
             << p.burstTime << "\t" << p.waitingTime << "\t" << p.turnaroundTime << "\n";
    }

    cout << "\nAverage Waiting Time: " << totalWaiting / n;
    cout << "\nAverage Turnaround Time: " << totalTurnaround / n << "\n";
}
```

```
// Function to calculate Round Robin Scheduling
void RoundRobin(vector<Process> processes, int quantum) {
    int n = processes.size();
    float totalWaiting = 0, totalTurnaround = 0;
    int currentTime = 0;
    queue<int> readyQueue;
    vector<int> gantt;

    // Sort by arrival time
    sort(processes.begin(), processes.end(), [](Process a, Process b) {
        return a.arrivalTime < b.arrivalTime;
    });

    for (int i = 0; i < n; i++)
        processes[i].remainingTime = processes[i].burstTime;

    readyQueue.push(0);
    vector<bool> inQueue(n, false);
    inQueue[0] = true;

    cout << "\n===== Round Robin (RR) Scheduling =====\n";
    cout << "Time Quantum = " << quantum << "\n";
    cout << "Gantt Chart: ";

    while (!readyQueue.empty()) {
        int idx = readyQueue.front();
        readyQueue.pop();

        if (currentTime < processes[idx].arrivalTime)
            currentTime = processes[idx].arrivalTime;

        gantt.push_back(processes[idx].pid);
```

```
        cout << "| P" << processes[idx].pid << " ";

        int execTime = min(quantum, processes[idx].remainingTime);
        processes[idx].remainingTime -= execTime;
        currentTime += execTime;

        // Add newly arrived processes
        for (int i = 0; i < n; i++) {
            if (!inQueue[i] && processes[i].arrivalTime <= currentTime &&
processes[i].remainingTime > 0) {
                readyQueue.push(i);
                inQueue[i] = true;
            }
        }

        // Re-add current process if not finished
        if (processes[idx].remainingTime > 0) {
            readyQueue.push(idx);
        } else {
            processes[idx].turnaroundTime = currentTime - processes[idx].arrivalTime;
            processes[idx].waitingTime = processes[idx].turnaroundTime -
processes[idx].burstTime;
            totalWaiting += processes[idx].waitingTime;
            totalTurnaround += processes[idx].turnaroundTime;
        }
    }

    cout << "|\n";
    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
    cout << "-----------------------------------------\n";
    for (auto &p : processes) {
        cout << "P" << p.pid << "\t" << p.arrivalTime << "\t"
```

```
            << p.burstTime << "\t" << p.waitingTime << "\t" << p.turnaroundTime << "\n";
    }

    cout << "\nAverage Waiting Time: " << totalWaiting / n;
    cout << "\nAverage Turnaround Time: " << totalTurnaround / n << "\n";
}

int main() {
    cout << "=== CPU Scheduling Algorithms: FCFS & Round Robin ===\n";

    // Example data
    vector<Process> processes = {
        {1, 5, 0},  // pid, burst, arrival
        {2, 3, 1},
        {3, 8, 2},
        {4, 6, 3}
    };

    cout << "\nInput Process Data:\n";
    cout << "Process\tArrival\tBurst\n";
    cout << "----------------------\n";
    for (auto &p : processes) {
        cout << "P" << p.pid << "\t" << p.arrivalTime << "\t" << p.burstTime << "\n";
    }

    FCFS(processes);

    int quantum = 3;
    RoundRobin(processes, quantum);

    cout << "\n=== Simulation Complete ===\n";
    return 0;
```

```
}
```

User Input Enabled

```
#include <bits/stdc++.h>
using namespace std;

struct Process {
    int pid;
    int burstTime;
    int arrivalTime;
    int waitingTime;
    int turnaroundTime;
    int remainingTime;
};

void FCFS(vector<Process> processes) {
    int n = processes.size();
    int currentTime = 0;
    float totalWaiting = 0, totalTurnaround = 0;

    sort(processes.begin(), processes.end(), [](Process a, Process b) {
        return a.arrivalTime < b.arrivalTime;
    });

    cout << "\n===== First Come First Serve (FCFS) Scheduling =====\n";
    cout << "Gantt Chart: ";

    for (int i = 0; i < n; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
```

```
        cout << "I P" << processes[i].pid << " ";
        processes[i].waitingTime = currentTime - processes[i].arrivalTime;
        currentTime += processes[i].burstTime;
        processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;

        totalWaiting += processes[i].waitingTime;
        totalTurnaround += processes[i].turnaroundTime;
    }

    cout << "I\n";
    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
    cout << "-------------------------------------------\n";
    for (auto &p : processes) {
        cout << "P" << p.pid << "\t" << p.arrivalTime << "\t"
             << p.burstTime << "\t" << p.waitingTime << "\t" << p.turnaroundTime << "\n";
    }

    cout << "\nAverage Waiting Time: " << totalWaiting / n;
    cout << "\nAverage Turnaround Time: " << totalTurnaround / n << "\n";
}

void RoundRobin(vector<Process> processes, int quantum) {
    int n = processes.size();
    float totalWaiting = 0, totalTurnaround = 0;
    int currentTime = 0;
    queue<int> readyQueue;
    vector<int> gantt;

    sort(processes.begin(), processes.end(), [](Process a, Process b) {
        return a.arrivalTime < b.arrivalTime;
    });
```

```
    for (int i = 0; i < n; i++)
        processes[i].remainingTime = processes[i].burstTime;

    readyQueue.push(0);
    vector<bool> inQueue(n, false);
    inQueue[0] = true;

    cout << "\n===== Round Robin (RR) Scheduling =====\n";
    cout << "Time Quantum = " << quantum << "\n";
    cout << "Gantt Chart: ";

    while (!readyQueue.empty()) {
        int idx = readyQueue.front();
        readyQueue.pop();

        if (currentTime < processes[idx].arrivalTime)
            currentTime = processes[idx].arrivalTime;

        gantt.push_back(processes[idx].pid);
        cout << "| P" << processes[idx].pid << " ";

        int execTime = min(quantum, processes[idx].remainingTime);
        processes[idx].remainingTime -= execTime;
        currentTime += execTime;

        for (int i = 0; i < n; i++) {
            if (!inQueue[i] && processes[i].arrivalTime <= currentTime &&
processes[i].remainingTime > 0) {
                readyQueue.push(i);
                inQueue[i] = true;
            }
        }
```

```
        if (processes[idx].remainingTime > 0) {
            readyQueue.push(idx);
        } else {
            processes[idx].turnaroundTime = currentTime - processes[idx].arrivalTime;
            processes[idx].waitingTime = processes[idx].turnaroundTime -
processes[idx].burstTime;
            totalWaiting += processes[idx].waitingTime;
            totalTurnaround += processes[idx].turnaroundTime;
        }
    }

    cout << "l\n";
    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
    cout << "-------------------------------------------\n";
    for (auto &p : processes) {
        cout << "P" << p.pid << "\t" << p.arrivalTime << "\t"
             << p.burstTime << "\t" << p.waitingTime << "\t" << p.turnaroundTime << "\n";
    }

    cout << "\nAverage Waiting Time: " << totalWaiting / n;
    cout << "\nAverage Turnaround Time: " << totalTurnaround / n << "\n";
}

int main() {
    cout << "=== CPU Scheduling Algorithms: FCFS & Round Robin ===\n";

    int n;
    cout << "\nEnter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
```

```
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "\nEnter Arrival Time for P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime;
        cout << "Enter Burst Time for P" << i + 1 << ": ";
        cin >> processes[i].burstTime;
    }

    cout << "\nInput Process Data:\n";
    cout << "Process\tArrival\tBurst\n";
    cout << "----------------------\n";
    for (auto &p : processes) {
        cout << "P" << p.pid << "\t" << p.arrivalTime << "\t" << p.burstTime << "\n";
    }

    FCFS(processes);

    int quantum;
    cout << "\nEnter Time Quantum for Round Robin: ";
    cin >> quantum;

    RoundRobin(processes, quantum);

    cout << "\n=== Simulation Complete ===\n";
    return 0;
}
```

Sample Input

```
Enter number of processes: 4
```

Enter Arrival Time for P1: 0

Enter Burst Time for P1: 5

Enter Arrival Time for P2: 1

Enter Burst Time for P2: 3

Enter Arrival Time for P3: 2

Enter Burst Time for P3: 8

Enter Arrival Time for P4: 3

Enter Burst Time for P4: 6

Enter Time Quantum for Round Robin: 3

6.Write a program to simulate memory placement strategies Best Fit ,First Fit , Next Fit , Worst Fit.

```
#include <bits/stdc++.h>
using namespace std;

// Structure to hold memory block info
struct Block {
    int id;
    int size;
    bool allocated;
};

// Structure to hold process info
struct Process {
    int id;
    int size;
```

```
};

// Utility function to print allocation result
void printResult(string method, vector<int> allocation, vector<Block> blocks,
vector<Process> processes) {
    cout << "\n=== " << method << " Allocation Result ===\n";
    cout << "Process ID\tProcess Size\tBlock Allocated\n";
    cout << "-----------------------------------------\n";
    for (int i = 0; i < processes.size(); i++) {
        cout << "P" << processes[i].id << "\t\t" << processes[i].size << "\t\t";
        if (allocation[i] != -1)
            cout << "Block " << allocation[i] + 1 << " (" << blocks[allocation[i]].size << ")";
        else
            cout << "Not Allocated";
        cout << "\n";
    }
}

// First Fit Algorithm
void firstFit(vector<Block> blocks, vector<Process> processes) {
    vector<int> allocation(processes.size(), -1);
    for (int i = 0; i < processes.size(); i++) {
        for (int j = 0; j < blocks.size(); j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                allocation[i] = j;
                blocks[j].allocated = true;
                blocks[j].size -= processes[i].size;
                break;
            }
        }
    }
    printResult("First Fit", allocation, blocks, processes);
```

```
}

// Best Fit Algorithm
void bestFit(vector<Block> blocks, vector<Process> processes) {
    vector<int> allocation(processes.size(), -1);
    for (int i = 0; i < processes.size(); i++) {
        int bestIdx = -1;
        int minWaste = INT_MAX;
        for (int j = 0; j < blocks.size(); j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                int waste = blocks[j].size - processes[i].size;
                if (waste < minWaste) {
                    minWaste = waste;
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blocks[bestIdx].allocated = true;
            blocks[bestIdx].size -= processes[i].size;
        }
    }
    printResult("Best Fit", allocation, blocks, processes);
}

// Worst Fit Algorithm
void worstFit(vector<Block> blocks, vector<Process> processes) {
    vector<int> allocation(processes.size(), -1);
    for (int i = 0; i < processes.size(); i++) {
        int worstIdx = -1;
        int maxWaste = -1;
```

```
        for (int j = 0; j < blocks.size(); j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                int waste = blocks[j].size - processes[i].size;
                if (waste > maxWaste) {
                    maxWaste = waste;
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blocks[worstIdx].allocated = true;
            blocks[worstIdx].size -= processes[i].size;
        }
    }
    printResult("Worst Fit", allocation, blocks, processes);
}

// Next Fit Algorithm
void nextFit(vector<Block> blocks, vector<Process> processes) {
    vector<int> allocation(processes.size(), -1);
    int lastAllocated = 0; // start searching from last allocated block

    for (int i = 0; i < processes.size(); i++) {
        int count = 0;
        bool allocatedFlag = false;
        while (count < blocks.size()) {
            int j = (lastAllocated + count) % blocks.size();
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                allocation[i] = j;
                blocks[j].allocated = true;
                blocks[j].size -= processes[i].size;
```

```
            lastAllocated = j; // next search starts from here
            allocatedFlag = true;
            break;
        }
        count++;
    }
    if (!allocatedFlag)
        allocation[i] = -1;
  }
  printResult("Next Fit", allocation, blocks, processes);
}

int main() {
  cout << "=== MEMORY PLACEMENT STRATEGIES SIMULATION ===\n";

  // Example memory blocks and processes
  vector<Block> blocks = {
    {1, 100, false},
    {2, 500, false},
    {3, 200, false},
    {4, 300, false},
    {5, 600, false}
  };

  vector<Process> processes = {
    {1, 212},
    {2, 417},
    {3, 112},
    {4, 426}
  };

  cout << "\nMemory Blocks:\n";
```

```
    for (auto b : blocks) cout << "Block " << b.id << ": " << b.size << " KB\n";
    cout << "\nProcesses:\n";
    for (auto p : processes) cout << "Process " << p.id << ": " << p.size << " KB\n";

    // Run all strategies
    firstFit(blocks, processes);
    bestFit(blocks, processes);
    worstFit(blocks, processes);
    nextFit(blocks, processes);

    cout << "\n=== Simulation Complete ===\n";
    return 0;
}
```

User Input Enabled

```
#include <bits/stdc++.h>
using namespace std;

struct Block {
    int id;
    int size;
    bool allocated;
};

struct Process {
    int id;
    int size;
};
```

```
void printResult(string method, vector<int> allocation, vector<Block> blocks,
vector<Process> processes) {
    cout << "\n=== " << method << " Allocation Result ===\n";
    cout << "Process ID\tProcess Size\tBlock Allocated\n";
    cout << "-------------------------------------------\n";
    for (int i = 0; i < processes.size(); i++) {
        cout << "P" << processes[i].id << "\t\t" << processes[i].size << "\t\t";
        if (allocation[i] != -1)
            cout << "Block " << blocks[allocation[i]].id << " (" << blocks[allocation[i]].size << ")";
        else
            cout << "Not Allocated";
        cout << "\n";
    }
}

void firstFit(vector<Block> blocks, vector<Process> processes) {
    vector<int> allocation(processes.size(), -1);
    for (int i = 0; i < processes.size(); i++) {
        for (int j = 0; j < blocks.size(); j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                allocation[i] = j;
                blocks[j].allocated = true;
                blocks[j].size -= processes[i].size;
                break;
            }
        }
    }
    printResult("First Fit", allocation, blocks, processes);
}

void bestFit(vector<Block> blocks, vector<Process> processes) {
    vector<int> allocation(processes.size(), -1);
```

```
    for (int i = 0; i < processes.size(); i++) {
        int bestIdx = -1;
        int minWaste = INT_MAX;
        for (int j = 0; j < blocks.size(); j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                int waste = blocks[j].size - processes[i].size;
                if (waste < minWaste) {
                    minWaste = waste;
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blocks[bestIdx].allocated = true;
            blocks[bestIdx].size -= processes[i].size;
        }
    }
    printResult("Best Fit", allocation, blocks, processes);
}

void worstFit(vector<Block> blocks, vector<Process> processes) {
    vector<int> allocation(processes.size(), -1);
    for (int i = 0; i < processes.size(); i++) {
        int worstIdx = -1;
        int maxWaste = -1;
        for (int j = 0; j < blocks.size(); j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                int waste = blocks[j].size - processes[i].size;
                if (waste > maxWaste) {
                    maxWaste = waste;
                    worstIdx = j;
```

```
            }
        }
    }
    if (worstIdx != -1) {
        allocation[i] = worstIdx;
        blocks[worstIdx].allocated = true;
        blocks[worstIdx].size -= processes[i].size;
    }
  }
  printResult("Worst Fit", allocation, blocks, processes);
}

void nextFit(vector<Block> blocks, vector<Process> processes) {
  vector<int> allocation(processes.size(), -1);
  int lastAllocated = 0;
  for (int i = 0; i < processes.size(); i++) {
    int count = 0;
    bool allocatedFlag = false;
    while (count < blocks.size()) {
      int j = (lastAllocated + count) % blocks.size();
      if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
        allocation[i] = j;
        blocks[j].allocated = true;
        blocks[j].size -= processes[i].size;
        lastAllocated = j;
        allocatedFlag = true;
        break;
      }
      count++;
    }
    if (!allocatedFlag)
      allocation[i] = -1;
```

```
    }
    printResult("Next Fit", allocation, blocks, processes);
}

int main() {
    cout << "=== MEMORY PLACEMENT STRATEGIES SIMULATION ===\n";

    int blockCount, processCount;
    cout << "\nEnter number of memory blocks: ";
    cin >> blockCount;

    vector<Block> blocks(blockCount);
    cout << "Enter sizes of each memory block:\n";
    for (int i = 0; i < blockCount; i++) {
        cout << "Block " << i + 1 << ": ";
        cin >> blocks[i].size;
        blocks[i].id = i + 1;
        blocks[i].allocated = false;
    }

    cout << "\nEnter number of processes: ";
    cin >> processCount;

    vector<Process> processes(processCount);
    cout << "Enter sizes of each process:\n";
    for (int i = 0; i < processCount; i++) {
        cout << "Process " << i + 1 << ": ";
        cin >> processes[i].size;
        processes[i].id = i + 1;
    }

    cout << "\nMemory Blocks:\n";
```

```
    for (auto b : blocks) cout << "Block " << b.id << ": " << b.size << " KB\n";

    cout << "\nProcesses:\n";
    for (auto p : processes) cout << "Process " << p.id << ": " << p.size << " KB\n";

    firstFit(blocks, processes);
    bestFit(blocks, processes);
    worstFit(blocks, processes);
    nextFit(blocks, processes);

    cout << "\n=== Simulation Complete ===\n";
    return 0;
}
```

Sample Input

```
Enter number of memory blocks: 5
Enter sizes of each memory block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600

Enter number of processes: 4
Enter sizes of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426
```

7.Write a program to simulate memory replacement Algorithm such as First In First Out (FIFO) , Optimal , Recently Used , Most Recently Used (MRU) , and random Page replacement.

```
#include <bits/stdc++.h>
using namespace std;

// Function to print current frames
void printFrames(vector<int>& frames) {
    cout << "[ ";
    for (int f : frames) cout << f << " ";
    cout << "]" << endl;
}

// FIFO Page Replacement
int fifo(vector<int> pages, int framesCount) {
    cout << "\n--- FIFO Page Replacement ---\n";
    vector<int> frames;
    queue<int> q;
    int pageFaults = 0;

    for (int page : pages) {
        cout << "Page " << page << ": ";
```

```
        if (find(frames.begin(), frames.end(), page) == frames.end()) {
            pageFaults++;
            if (frames.size() == framesCount) {
                int old = q.front(); q.pop();
                frames.erase(remove(frames.begin(), frames.end(), old), frames.end());
            }
            frames.push_back(page);
            q.push(page);
            cout << "(Fault) ";
        } else {
            cout << "(Hit)   ";
        }
        printFrames(frames);
    }
    cout << "Total Page Faults (FIFO): " << pageFaults << "\n";
    return pageFaults;
}

// Optimal Page Replacement
int optimal(vector<int> pages, int framesCount) {
    cout << "\n--- Optimal Page Replacement ---\n";
    vector<int> frames;
    int pageFaults = 0;

    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        cout << "Page " << page << ": ";
        if (find(frames.begin(), frames.end(), page) == frames.end()) {
            pageFaults++;
            if (frames.size() == framesCount) {
                int farthest = i, idx = -1;
                for (int j = 0; j < frames.size(); j++) {
```

```
                int nextUse = INT_MAX;
                for (int k = i + 1; k < pages.size(); k++) {
                    if (pages[k] == frames[j]) { nextUse = k; break; }
                }
                if (nextUse > farthest) {
                    farthest = nextUse;
                    idx = j;
                }
            }
            frames[idx] = page;
        } else frames.push_back(page);
        cout << "(Fault) ";
    } else cout << "(Hit)   ";
    printFrames(frames);
  }
  cout << "Total Page Faults (Optimal): " << pageFaults << "\n";
  return pageFaults;
}

// LRU Page Replacement
int lru(vector<int> pages, int framesCount) {
  cout << "\n--- Least Recently Used (LRU) ---\n";
  vector<int> frames;
  unordered_map<int, int> recent; // page -> last used index
  int pageFaults = 0;

  for (int i = 0; i < pages.size(); i++) {
    int page = pages[i];
    cout << "Page " << page << ": ";
    if (find(frames.begin(), frames.end(), page) == frames.end()) {
      pageFaults++;
      if (frames.size() == framesCount) {
```

```
            int lruPage = frames[0], minIndex = recent[frames[0]];
            for (int p : frames) {
                if (recent[p] < minIndex) {
                    minIndex = recent[p];
                    lruPage = p;
                }
            }
            frames.erase(remove(frames.begin(), frames.end(), lruPage), frames.end());
        }
        frames.push_back(page);
        cout << "(Fault) ";
    } else cout << "(Hit)   ";
    recent[page] = i;
    printFrames(frames);
  }
  cout << "Total Page Faults (LRU): " << pageFaults << "\n";
  return pageFaults;
}

// MRU Page Replacement
int mru(vector<int> pages, int framesCount) {
  cout << "\n--- Most Recently Used (MRU) ---\n";
  vector<int> frames;
  unordered_map<int, int> recent;
  int pageFaults = 0;

  for (int i = 0; i < pages.size(); i++) {
    int page = pages[i];
    cout << "Page " << page << ": ";
    if (find(frames.begin(), frames.end(), page) == frames.end()) {
      pageFaults++;
      if (frames.size() == framesCount) {
```

```
            int mruPage = frames[0], maxIndex = recent[frames[0]];
            for (int p : frames) {
                if (recent[p] > maxIndex) {
                    maxIndex = recent[p];
                    mruPage = p;
                }
            }
            frames.erase(remove(frames.begin(), frames.end(), mruPage), frames.end());
        }
        frames.push_back(page);
        cout << "(Fault) ";
    } else cout << "(Hit)   ";
    recent[page] = i;
    printFrames(frames);
  }
  cout << "Total Page Faults (MRU): " << pageFaults << "\n";
  return pageFaults;
}

// Random Page Replacement
int randomReplacement(vector<int> pages, int framesCount) {
  cout << "\n--- Random Page Replacement ---\n";
  vector<int> frames;
  int pageFaults = 0;
  srand(time(0));

  for (int page : pages) {
    cout << "Page " << page << ": ";
    if (find(frames.begin(), frames.end(), page) == frames.end()) {
      pageFaults++;
      if (frames.size() == framesCount) {
        int idx = rand() % framesCount;
```

```
                frames[idx] = page;
            } else frames.push_back(page);
            cout << "(Fault) ";
        } else cout << "(Hit)   ";
        printFrames(frames);
    }
    cout << "Total Page Faults (Random): " << pageFaults << "\n";
    return pageFaults;
}

int main() {
    cout << "=== PAGE REPLACEMENT ALGORITHMS SIMULATION ===\n";

    // Example reference string
    vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int framesCount = 3;

    cout << "\nReference String: ";
    for (int p : pages) cout << p << " ";
    cout << "\nNumber of Frames: " << framesCount << "\n";

    fifo(pages, framesCount);
    optimal(pages, framesCount);
    lru(pages, framesCount);
    mru(pages, framesCount);
    randomReplacement(pages, framesCount);

    return 0;
}
```

Input Driven

```
 #include <bits/stdc++.h>
using namespace std;

void printFrames(vector<int>& frames) {
    cout << "[ ";
    for (int f : frames) cout << f << " ";
    cout << "]" << endl;
}

int fifo(vector<int> pages, int framesCount) {
    cout << "\n--- FIFO Page Replacement ---\n";
    vector<int> frames;
    queue<int> q;
    int pageFaults = 0;

    for (int page : pages) {
        cout << "Page " << page << ": ";
        if (find(frames.begin(), frames.end(), page) == frames.end()) {
            pageFaults++;
            if (frames.size() == framesCount) {
                int old = q.front(); q.pop();
                frames.erase(remove(frames.begin(), frames.end(), old), frames.end());
            }
            frames.push_back(page);
            q.push(page);
            cout << "(Fault) ";
        } else {
            cout << "(Hit)   ";
        }
        printFrames(frames);
```

```
    }
    cout << "Total Page Faults (FIFO): " << pageFaults << "\n";
    return pageFaults;
}

int optimal(vector<int> pages, int framesCount) {
    cout << "\n--- Optimal Page Replacement ---\n";
    vector<int> frames;
    int pageFaults = 0;

    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        cout << "Page " << page << ": ";
        if (find(frames.begin(), frames.end(), page) == frames.end()) {
            pageFaults++;
            if (frames.size() == framesCount) {
                int farthest = i, idx = -1;
                for (int j = 0; j < frames.size(); j++) {
                    int nextUse = INT_MAX;
                    for (int k = i + 1; k < pages.size(); k++) {
                        if (pages[k] == frames[j]) { nextUse = k; break; }
                    }
                    if (nextUse > farthest) {
                        farthest = nextUse;
                        idx = j;
                    }
                }
                frames[idx] = page;
            } else frames.push_back(page);
            cout << "(Fault) ";
        } else cout << "(Hit)   ";
        printFrames(frames);
```

```
    }
    cout << "Total Page Faults (Optimal): " << pageFaults << "\n";
    return pageFaults;
}

int lru(vector<int> pages, int framesCount) {
    cout << "\n--- Least Recently Used (LRU) ---\n";
    vector<int> frames;
    unordered_map<int, int> recent;
    int pageFaults = 0;

    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        cout << "Page " << page << ": ";
        if (find(frames.begin(), frames.end(), page) == frames.end()) {
            pageFaults++;
            if (frames.size() == framesCount) {
                int lruPage = frames[0], minIndex = recent[frames[0]];
                for (int p : frames) {
                    if (recent[p] < minIndex) {
                        minIndex = recent[p];
                        lruPage = p;
                    }
                }
                frames.erase(remove(frames.begin(), frames.end(), lruPage), frames.end());
            }
            frames.push_back(page);
            cout << "(Fault) ";
        } else cout << "(Hit)   ";
        recent[page] = i;
        printFrames(frames);
    }
```

```
    cout << "Total Page Faults (LRU): " << pageFaults << "\n";
    return pageFaults;
}

int mru(vector<int> pages, int framesCount) {
    cout << "\n--- Most Recently Used (MRU) ---\n";
    vector<int> frames;
    unordered_map<int, int> recent;
    int pageFaults = 0;

    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        cout << "Page " << page << ": ";
        if (find(frames.begin(), frames.end(), page) == frames.end()) {
            pageFaults++;
            if (frames.size() == framesCount) {
                int mruPage = frames[0], maxIndex = recent[frames[0]];
                for (int p : frames) {
                    if (recent[p] > maxIndex) {
                        maxIndex = recent[p];
                        mruPage = p;
                    }
                }
                frames.erase(remove(frames.begin(), frames.end(), mruPage), frames.end());
            }
            frames.push_back(page);
            cout << "(Fault) ";
        } else cout << "(Hit)   ";
        recent[page] = i;
        printFrames(frames);
    }
    cout << "Total Page Faults (MRU): " << pageFaults << "\n";
```

```
    return pageFaults;
}

int randomReplacement(vector<int> pages, int framesCount) {
    cout << "\n--- Random Page Replacement ---\n";
    vector<int> frames;
    int pageFaults = 0;
    srand(time(0));

    for (int page : pages) {
        cout << "Page " << page << ": ";
        if (find(frames.begin(), frames.end(), page) == frames.end()) {
            pageFaults++;
            if (frames.size() == framesCount) {
                int idx = rand() % framesCount;
                frames[idx] = page;
            } else frames.push_back(page);
            cout << "(Fault) ";
        } else cout << "(Hit)   ";
        printFrames(frames);
    }
    cout << "Total Page Faults (Random): " << pageFaults << "\n";
    return pageFaults;
}

int main() {
    cout << "=== PAGE REPLACEMENT ALGORITHMS SIMULATION ===\n";

    int n, framesCount;
    cout << "\nEnter number of pages: ";
    cin >> n;
```

```
    vector<int> pages(n);
    cout << "Enter reference string: ";
    for (int i = 0; i < n; i++) cin >> pages[i];

    cout << "Enter number of frames: ";
    cin >> framesCount;

    cout << "\nReference String: ";
    for (int p : pages) cout << p << " ";
    cout << "\nNumber of Frames: " << framesCount << "\n";

    fifo(pages, framesCount);
    optimal(pages, framesCount);
    lru(pages, framesCount);
    mru(pages, framesCount);
    randomReplacement(pages, framesCount);

    return 0;
}
```

Sample Input

```
Enter number of pages: 13
Enter reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2
Enter number of frames: 3
```