```
/*Represent polynomial as a circularly linked list and
write a menu driven program to perform addition
and evaluation .*/


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

// Structure of a node
// in a circular linked list
struct Node {

      // Stores coefficient
      // of a node
      int coeff;

      // Stores power of'
      // variable x of a node
      int powx;

      // Stores power of
      // variable y of a node
      int powy;

      // Stores pointer
      // to next node
      struct Node* next;
};

// Function to dynamically create a node
void create_node(int c, int p1, int p2,
                                    struct Node** temp)
{

      // Stores new node
      struct Node *r;


      // Stores temp node
      struct Node *z
                  = *temp;


      // Dynamically create a new node
      r = (struct Node*)malloc(
                        sizeof(struct Node));


      // Update coefficient
      // of r
      r->coeff = c;


      // Update power of
      // variable x in r
      r->powx = p1;
      // Update power of
      // variable y in r
      r->powy = p2;


      // If z is null
      if (z == NULL) {


            // Update temp node
            (*temp) = r;


            // Update next pointer
            // of temp node
            (*temp)->next = (*temp);
      }
      else {


            // Update next pointer
            // of z
            r->next = z->next;


            // Update next pointer
            // of z
            z->next = r;


            // Update temp Node
            (*temp) = r;
      }
}


// Function to add polynomial of two list
void add_poly(struct Node* poly1,
      struct Node* poly2, struct Node** temp)
{


      // Stores head node of polynomial1
      struct Node *start1 = poly1;


      // Stores head node of polynomial1
      struct Node *start2 = poly2;


      // Update poly1
      poly1 = poly1->next;


      // Update poly2
      poly2 = poly2->next;


      // Traverse both circular linked list
```

```
while ((poly1 != start1 &&
                        poly2 != start2)) {


      // Stores new node
      struct Node* r;


      // Stores temp node
      struct Node* z
                  = *temp;


      // Dynamically create a new node
      r = (struct Node*)malloc(
                        sizeof(struct Node));


      // Update coefficient of r
      r->coeff = 0;


      // If power of x of poly1 is
      // greater than power of x of poly2
      if (poly1->powx > poly2->powx) {


            // Update coefficient of r
            r->coeff = poly1->coeff;


            // Update of power of x in r
            r->powx = poly1->powx;


            // Update of power of y in r
            r->powy = poly1->powy;


            // Update poly1
            poly1 = poly1->next;
      }


      // If power of x of 1st polynomial is
      // less than power of x of 2nd poly
      else if (poly1->powx < poly2->powx) {


            // Update coefficient OF r
            r->coeff = poly2->coeff;


            // Update power of x in r
            r->powx = poly2->powx;


            // Update power of y in r
            r->powy = poly2->powy;
            // Update ploy2
            poly2 = poly2->next;
      }


      // If power of x of 1st polynomial is
      // equal to power of x of 2nd poly
      else {


            // Power of y of 1st polynomial is
            // greater than power of y of poly2
            if (poly1->powy > poly2->powy) {


                  // Update coefficient of r
                  r->coeff = poly1->coeff;


                  // Update power of x in r
                  r->powx = poly1->powx;


                  // Update power of y in r
                  r->powy = poly1->powy;


                  // Update poly1
                  poly1 = poly1->next;
            }


            // If power of y of poly1 is
            // less than power of y of ploy2
            else if (poly1->powy <
                                    poly2->powy) {


                  // Update coefficient of r
                  r->coeff = poly2->coeff;


                  // Update power of x in r
                  r->powx = poly2->powx;


                  // Update power of y in r
                  r->powy = poly2->powy;


                  // Update poly2
                  poly2 = poly2->next;
            }


            // If power of y of 1st poly is
            // equal to power of y of ploy2
```

```
            else {

                  // Update coefficient of r
                  r->coeff = poly2->coeff
                              + poly1->coeff;

                  // Update power of x in r
                  r->powx = poly1->powx;

                  // Update power of y in r
                  r->powy = poly1->powy;

                  // Update poly1
                  poly1 = poly1->next;

                  // Update poly2
                  poly2 = poly2->next;
            }
      }

      // If z is null
      if (z == NULL) {

            // Update temp
            (*temp) = r;

            // Update next pointer
            // of temp
            (*temp)->next = (*temp);
      }
      else {

            // Update next pointer
            // of r
            r->next = z->next;

            // Update next pointer
            // of z
            z->next = r;

            // Update temp
            (*temp) = r;
      }
}

// If there are nodes left to be
// traversed in poly1 or poly2 then
// append them in resultant polynomial .
while (poly1 != start1 ||
                        poly2 != start2) {



      // If poly1 is not empty
      if (poly1 != start1) {


            // Stores new node
            struct Node *r;


            // Stores temp node
            struct Node *z = *temp;


            // Create new node
            r = (struct Node*)malloc(
                              sizeof(struct Node));


            // Update coefficient or r
            r->coeff = poly1->coeff;


            // Update power of x in r
            r->powx = poly1->powx;


            // Update power of y in r
            r->powy = poly1->powy;


            // Update poly1
            poly1 = poly1->next;


            // If z is null
            if (z == NULL) {


                  // Update temp
                  (*temp) = r;


                  // Update pointer
                  // to next node
                  (*temp)->next = (*temp);
            }
            else {


                  // Update next pointer
                  // of r
                  r->next = z->next;
            // Update next pointer of z
            z->next = r;

            // Update temp
            (*temp) = r;
        }
    }

    // If poly2 is not empty
    if (poly2 != start2) {

        // Stores new node
        struct Node *r;

        // Stores temp node
        struct Node *z = *temp;

        // Create new node
        r = (struct Node*)malloc(
                    sizeof(struct Node));

        // Update coefficient of z
        z->coeff = poly2->coeff;

        // Update power of x in z
        z->powx = poly2->powx;

        // Update power of y in z
        z->powy = poly2->powy;

        // Update poly2
        poly2 = poly2->next;

        // If z is null
        if (z == NULL) {

            // Update temp
            (*temp) = r;

            // Update next pointer
            // of temp
            (*temp)->next = (*temp);
        }
        else {

            // Update next pointer
```

```
                  // of r
                  r->next = z->next;


                  // Update next pointer
                  // of z
                  z->next = r;


                  // Update temp
                  (*temp) = r;
            }
      }
}


// Stores new node
struct Node *r;



// Stores temp node
struct Node *z = *temp;


// Create new node
r = (struct Node*)malloc(
      sizeof(struct Node));


// Update coefficient of r
r->coeff = 0;


// If power of x of start1 greater than
// power of x of start2
if (start1->powx > start2->powx) {


      // Update coefficient of r
      r->coeff = start1->coeff;


      // Update power of x in r
      r->powx = start1->powx;


      // Update power of y in r
      r->powy = start1->powy;
}


// If power of x of start1 less than
// power of x of start2
else if (start1->powx < start2->powx) {


      // Update coefficient of r
      r->coeff = start2->coeff;
      // Update power of x in r
      r->powx = start2->powx;


      // Update power of y in r
      r->powy = start2->powy;
}


// If power of x of start1 equal to
// power of x of start2
else {


      // If power of y of start1 greater than
      // power of y of start2
      if (start1->powy > start2->powy) {


            // Update coefficient of r
            r->coeff = start1->coeff;


            // Update power of x in r
            r->powx = start1->powx;


            // Update power of y in r
            r->powy = start1->powy;
      }


      // If power of y of start1 less than
      // power of y of start2
      else if (start1->powy <
                             start2->powy) {


            // Update coefficient of r
            r->coeff = start2->coeff;


            // Update power of x in r
            r->powx = start2->powx;


            // Update power of y in r
            r->powy = poly2->powy;
      }


      // If power of y of start1 equal to
      // power of y of start2
      else {


            // Update coefficient of r
```

```
                  r->coeff = start2->coeff
                                    + start1->coeff;


                  // Update power of x in r
                  r->powx = start1->powx;


                  // Update power of y in r
                  r->powy = start1->powy;
            }
      }


      // If z is null
      if (z == NULL) {


            // Update temp
            (*temp) = r;


            // Update next pointer
            // of temp
            (*temp)->next = (*temp);
      }
      else {


            // Update next pointer of r
            r->next = z->next;


            // Update next pointer of z
            z->next = r;


            // Update temp
            (*temp) = r;
      }
}


// Display the circular linked list
void display(struct Node* node)
{

      // Stores head node of list
      struct Node* start = node;


      // Update node
      node = node->next;


      // Traverse the list
      while (node != start &&
                  node->coeff != 0) {
```

```
            // Print coefficient of
            // current node
            cout << node->coeff;


            // If power of variable x
            // is not zero
            if (node->powx != 0)
                  cout << "x^" << node->powx;


            // If power of variable x
            // and y is not zero
            if(node->powx != 0 &&
                        node->powy != 0)
                  cout<<" * ";


            // If power of variable y
            // is not zero
            if (node->powy != 0)
                  cout << "y^" << node->powy;


            // Add next term of
            // the polynomial
            if (node != start &&
            node->next->coeff != 0) {
                  cout << " + ";
            }


            // Update node
            node = node->next;
      }


      // Print coefficient of
      // current node
      cout << node->coeff;


      // If power of variable x
      // is not zero
      if (node->powx != 0)
            cout << "x^" << node->powx;


      // If power of variable y
      // is not zero
      if (node->powy != 0)
            cout << "y^" << node->powy;
      cout << "\n\n";
}


// Driver Code
int main()
```

```
{

      // Stores node of
      // first polynomial
      struct Node *poly1 = NULL;


      // Stores node of
      // second polynomial
      struct Node *poly2 = NULL;


      // Stores node of resultant
      // polynomial
      struct Node *store = NULL;


      // Create first polynomial
      create_node(5, 2, 1, &poly1);
      create_node(4, 1, 2, &poly1);
      create_node(3, 1, 1, &poly1);
      create_node(2, 1, 0, &poly1);
      create_node(3, 0, 1, &poly1);
      create_node(2, 0, 0, &poly1);


      // Create second polynomial
      create_node(3, 1, 2, &poly2);
      create_node(4, 1, 0, &poly2);
      create_node(2, 0, 1, &poly2);
      create_node(6, 0, 0, &poly2);


      // Function call to add
      // two polynomial
      add_poly(poly1, poly2, &store);

      // Display polynomial 1
      cout << "Polynomial 1"
            << "\n";
      display(poly1);

      // Display polynomial 2
      cout << "Polynomial 2"
            << "\n";
      display(poly2);

      // Display final addition of 2-variable polynomial
      cout << "Polynomial after addition"
            << "\n";
      display(store);

      return 0;
}
```