```c
// ***
// *** You MUST modify this file
// ***

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "list.h"
#include "convert.h"

// DO NOT MODIFY FROM HERE --->>>
const int Operations[] = {'+', '-', '*', '(', ')'};

// return -1 if the word is not an operator
// return 0 if the word contains '+'
// return 1 if the word contains '-'
// return 2 if the word contains '*'
// return 3 if the word contains '('
// return 4 if the word contains ')'

int isOperator(char * word)
{
  int ind;
  int numop = sizeof(Operations) / sizeof(int);
  for (ind = 0; ind < numop; ind ++)
    {
    char *loc = strchr(word, Operations[ind]);
    if (loc != NULL && !isdigit(loc[1]))
        {
          return ind;
        }
    }
  return -1;
}
// <<<--- UNTIL HERE

// ***
// *** You MUST modify the convert function
// ***
#ifdef TEST_CONVERT
bool convert(List * arithlist)
{
  ListNode * pntr = (arithlist -> head);
  int val; //return value from isOperator
  int temp; //
  int nodeWord;
  List * output; //temp list that holds the operands
  List * operators; //temp list that holds the operators
  ListNode * p;
  output = malloc(sizeof(List));
  output -> head = NULL;
  output -> tail = NULL;

  operators = malloc(sizeof(List));
  operators -> head = NULL;
  operators -> tail = NULL;


  if (arithlist == NULL)
    {
      return true;
    }
  if ((arithlist -> head) == NULL)
    {
      return true;
    }

  while(pntr != NULL)
  {
   val = isOperator(pntr -> word);

   if(operators -> tail != NULL)
   {
    temp = isOperator(operators -> tail -> word);
   }

   if(operators -> tail == NULL)
   {
    temp == 5; //higher number than any precendence returned by
isOperator
   }

   if(val == -1)
   {
    addNode(output, pntr -> word);
   }

   else
   {
    if(operators -> tail == NULL)
    {
     addNode(operators, (pntr -> word));
    }

    else
    {
    switch (val)
    {
     case 0: //(+)
```

```
         if(temp >= 0 && temp != 3) //lower precedence
        {
         while(operators -> tail != NULL )
         {

          if((operators -> tail) != NULL && isOperator(operators ->
tail -> word) >= 0  && isOperator(operators -> tail -> word) != 3)

          {
           addNode(output, (operators -> tail -> word));
           deleteNode(operators, (operators -> tail));
          }
          else
          {
           break;
          }
         }
        }

         addNode(operators, (pntr -> word)); //adds pntr to operator
stack
         break;

      case 1: //(-)
         if(temp >= 0 && temp != 3)
         {
          while(operators -> tail != NULL )
         {
          if((operators -> tail) != NULL && isOperator(operators ->
tail -> word) >= 0  && isOperator(operators -> tail -> word) != 3)

          {
           addNode(output, (operators -> tail -> word));
           deleteNode(operators, (operators -> tail));
          }
          else
          {
           break;
          }
         }
         }

         addNode(operators, (pntr -> word));
         break;

      case 2: //(*)
         if(temp == 2 && temp != 3)
         {
         while(operators -> tail != NULL )
         {
          if((operators -> tail) != NULL && isOperator(operators ->
tail -> word) == 2  && isOperator(operators -> tail -> word) != 3)
          {
           addNode(output, (operators -> tail -> word));
           deleteNode(operators, (operators -> tail));
          }
          else
          {
           break;
          }
         }
        }

        addNode(operators, (pntr -> word));
        break;

     case 3: //('(')
        addNode(operators, (pntr -> word));
        break;

     case 4: //(')')
        p = operators -> tail;
        nodeWord = isOperator(p -> word);

        while(nodeWord != 3)
        {
         addNode(output, (p -> word));
         ListNode * holder = p;
         p = p -> prev;
         deleteNode(operators, holder);
         nodeWord = isOperator(p -> word);
        }

        deleteNode(operators, p);
        break;
    }
    }//switch
   }//else


   pntr = pntr -> next;
  }//while

  ListNode * temporary = operators -> tail;

  while(temporary != NULL)
  {
   addNode(output, temporary -> word);
   if(temporary != operators -> head)
   {
   temporary = temporary -> prev;
   free(temporary -> next);
   }

   else
   {
   free(temporary);
   break;
   }//else
  }//while


  ListNode * pointer;
  pointer = arithlist -> head;
  while(pointer != NULL)
  {
   ListNode * placeholder;
   placeholder = pointer;
   pointer = pointer -> next;
   free(placeholder);
  }

  arithlist -> head = output -> head;
  arithlist -> tail = output -> tail;

  free(output);
  free(operators);

  return true;
}
#endif
```