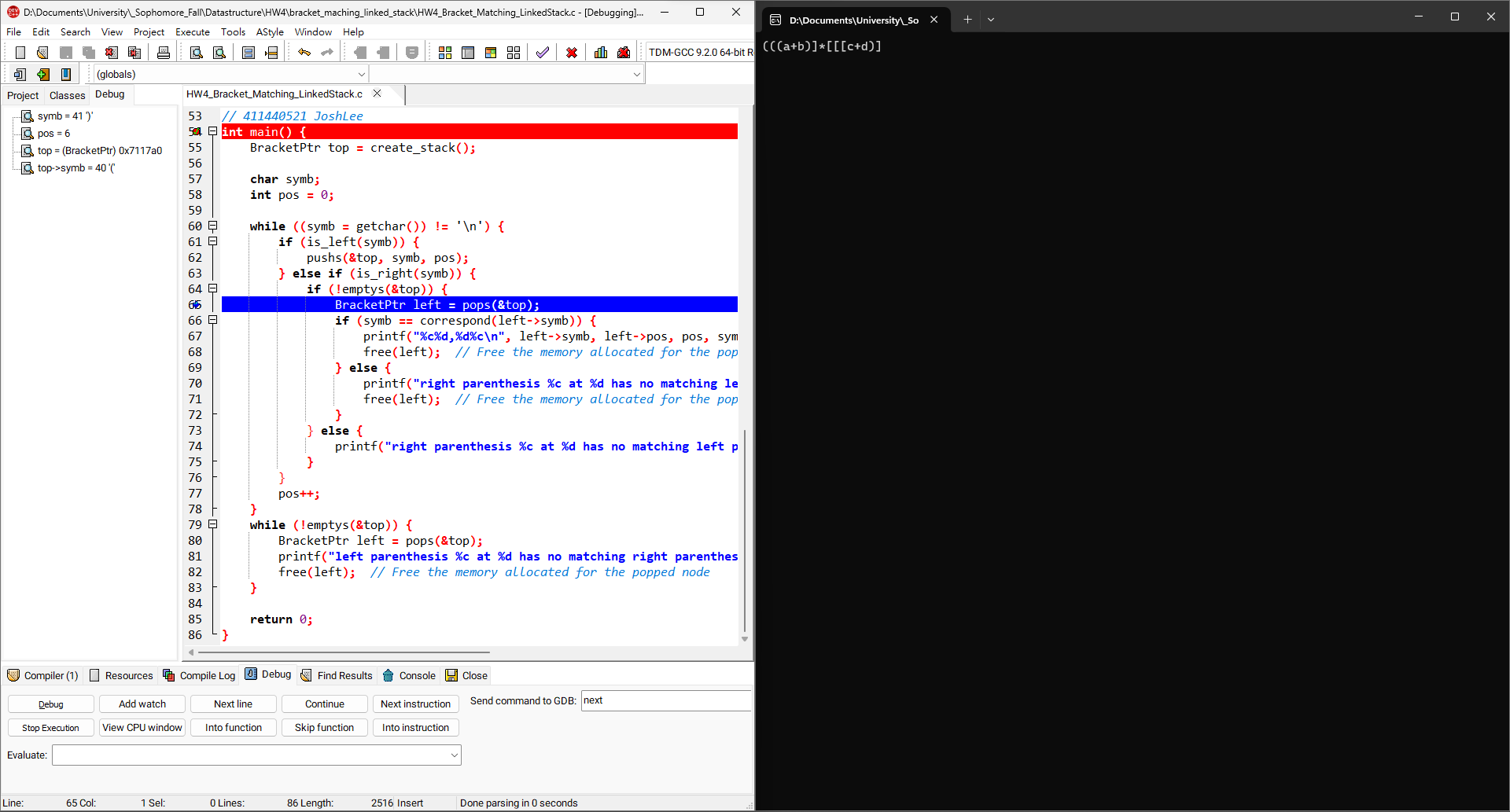
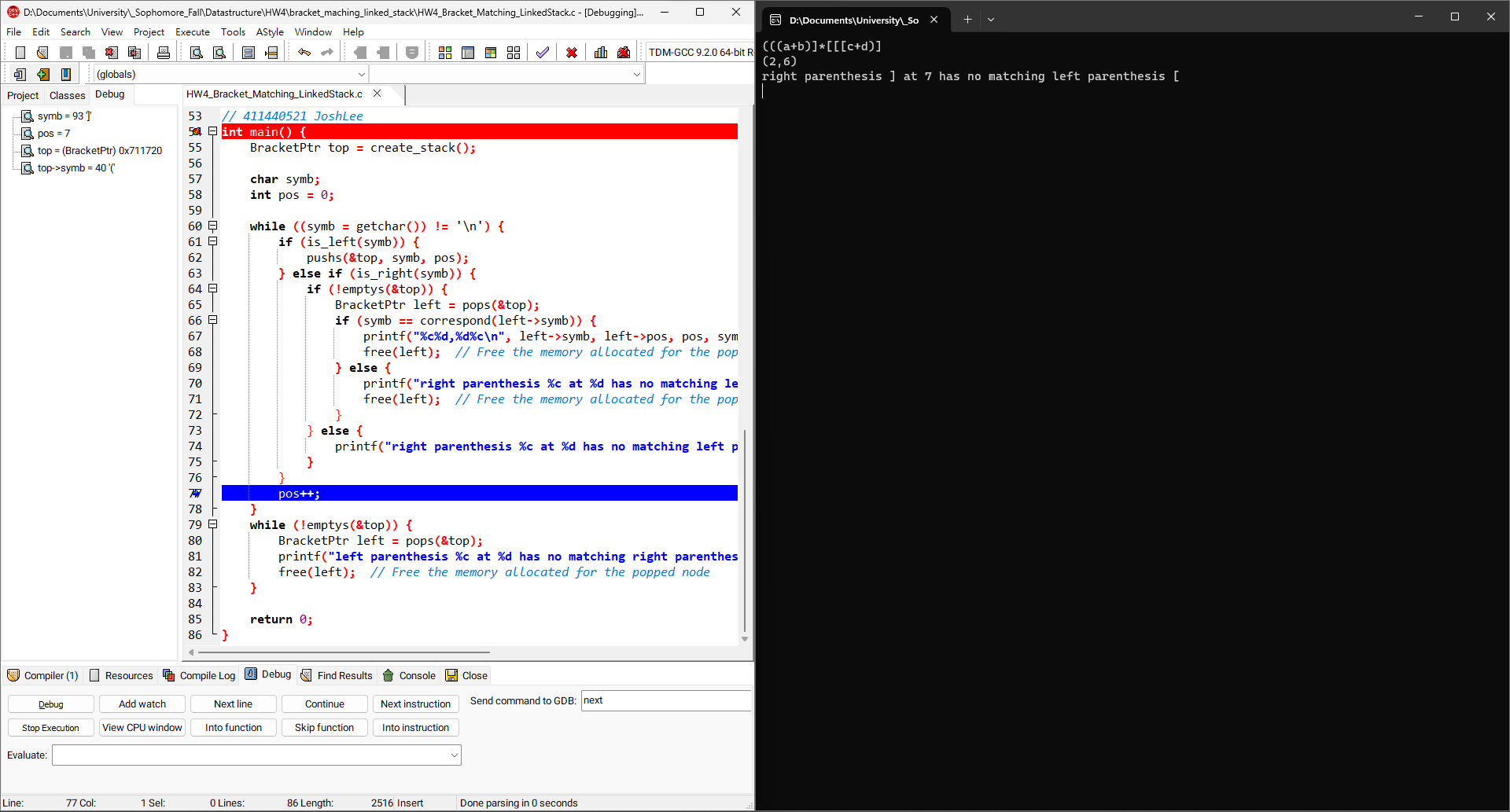
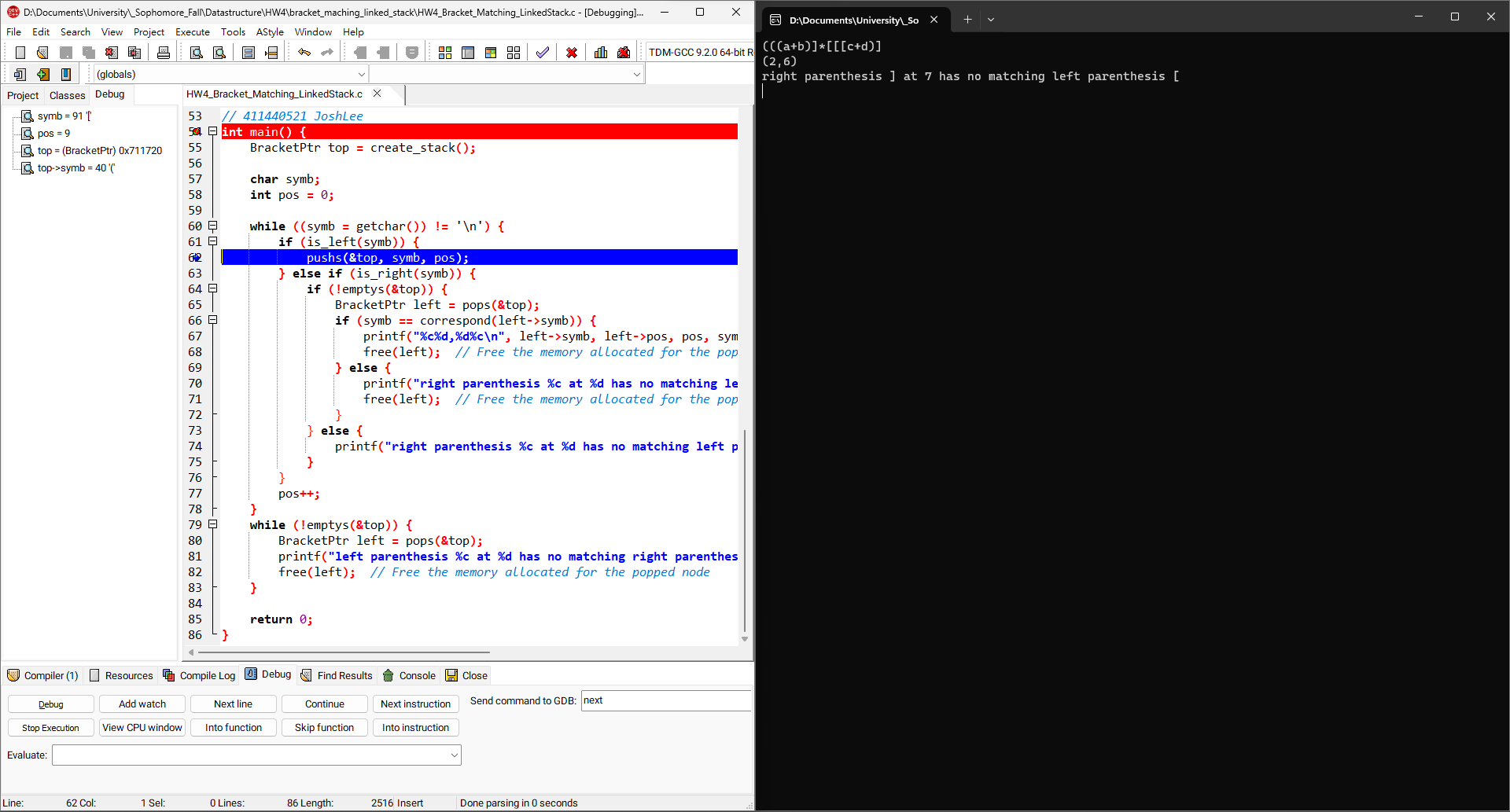
1

2

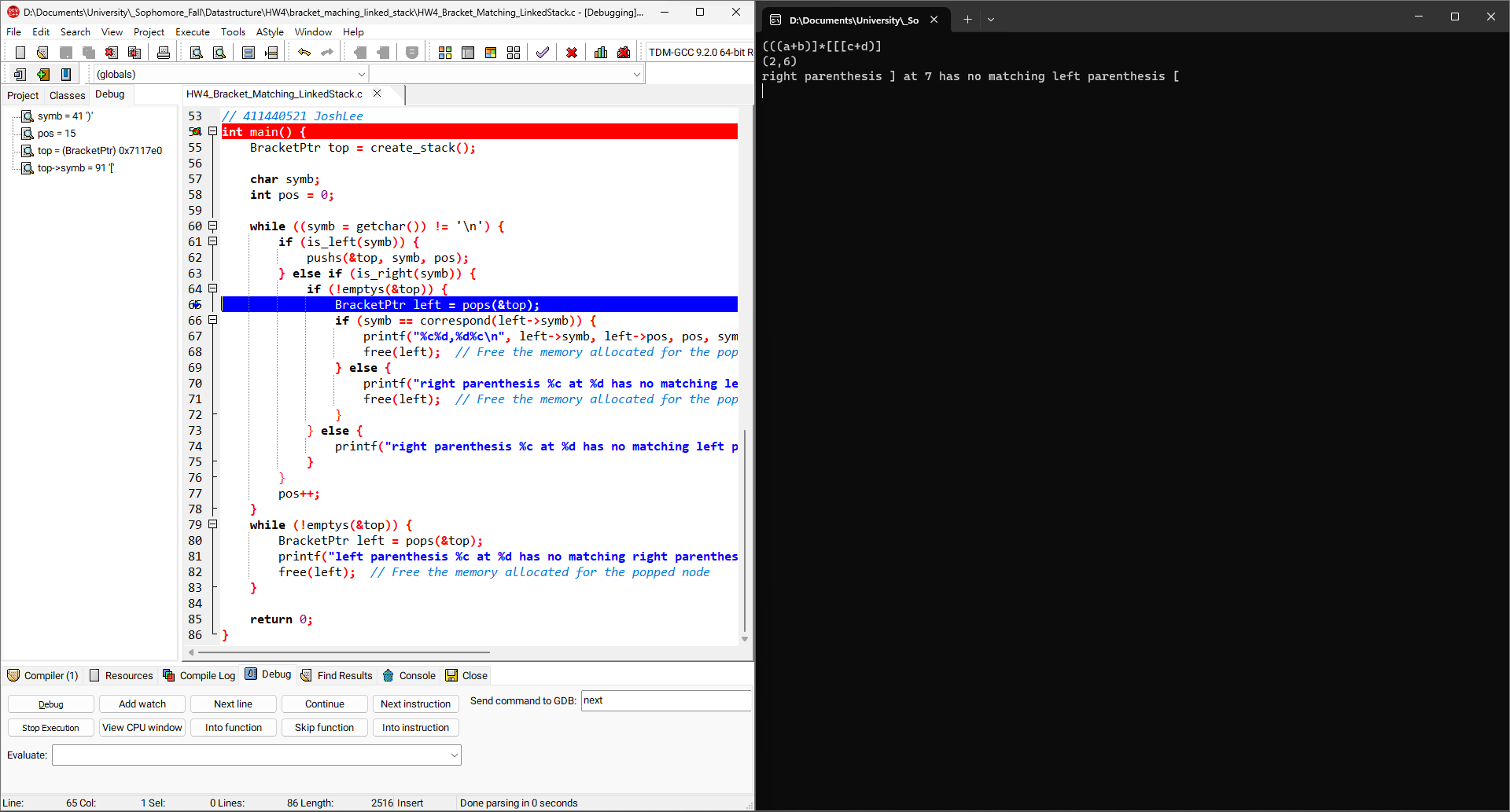
Push()重複三次  
(((

3

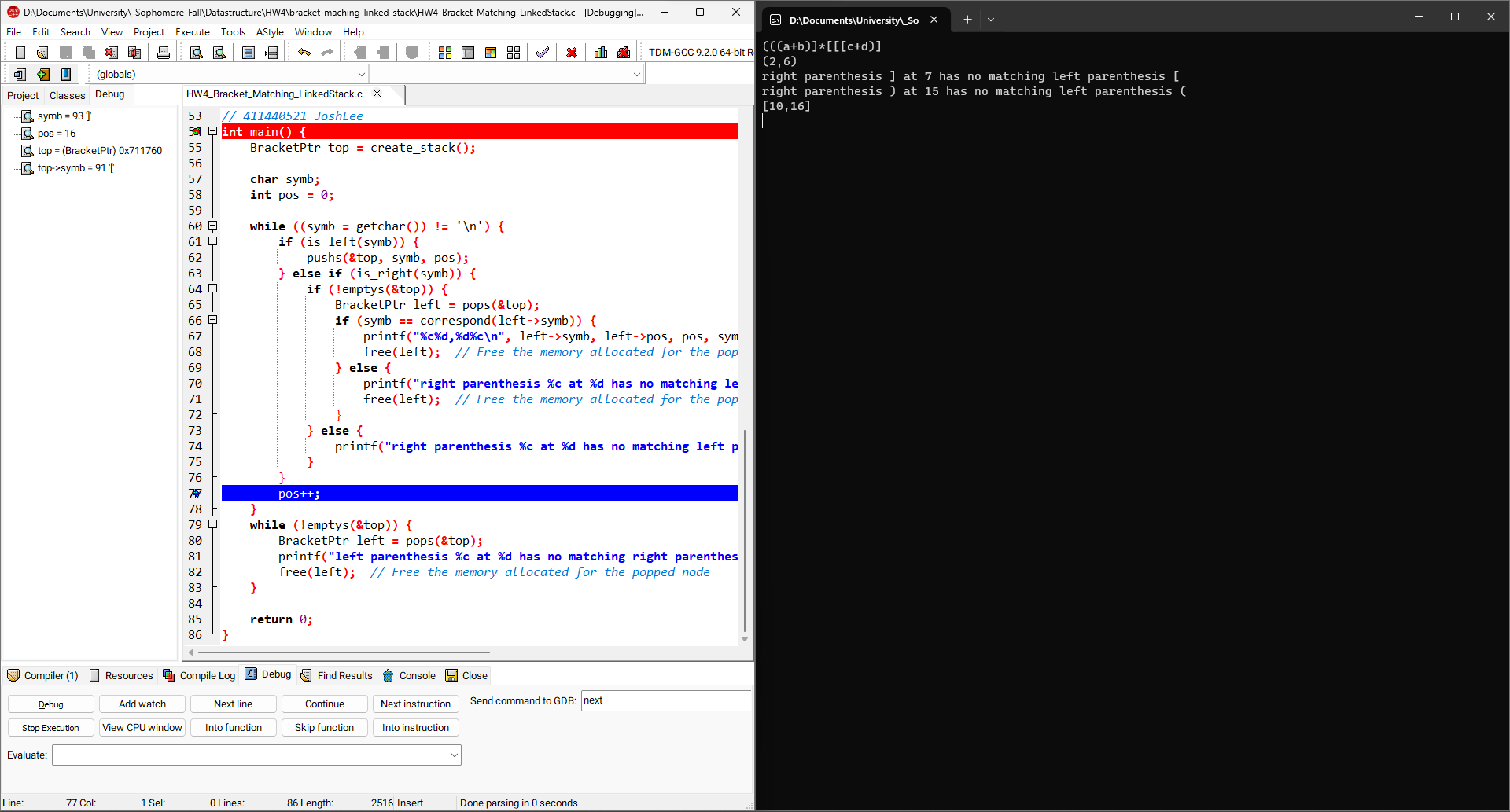
Pop()重複兩次  
(

4

Pop()兩次結果  
(

5

Push()三次  
([[[

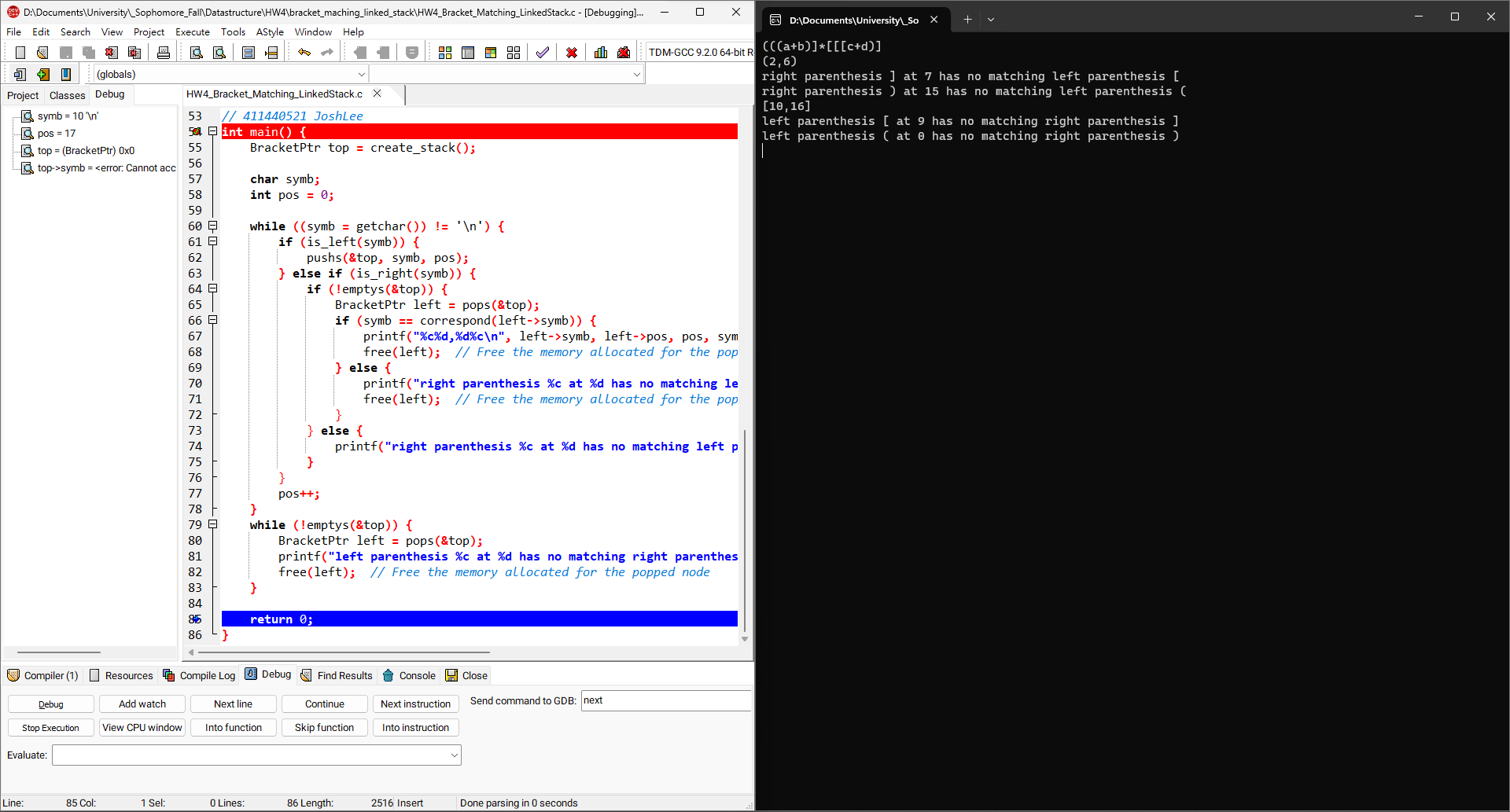
6

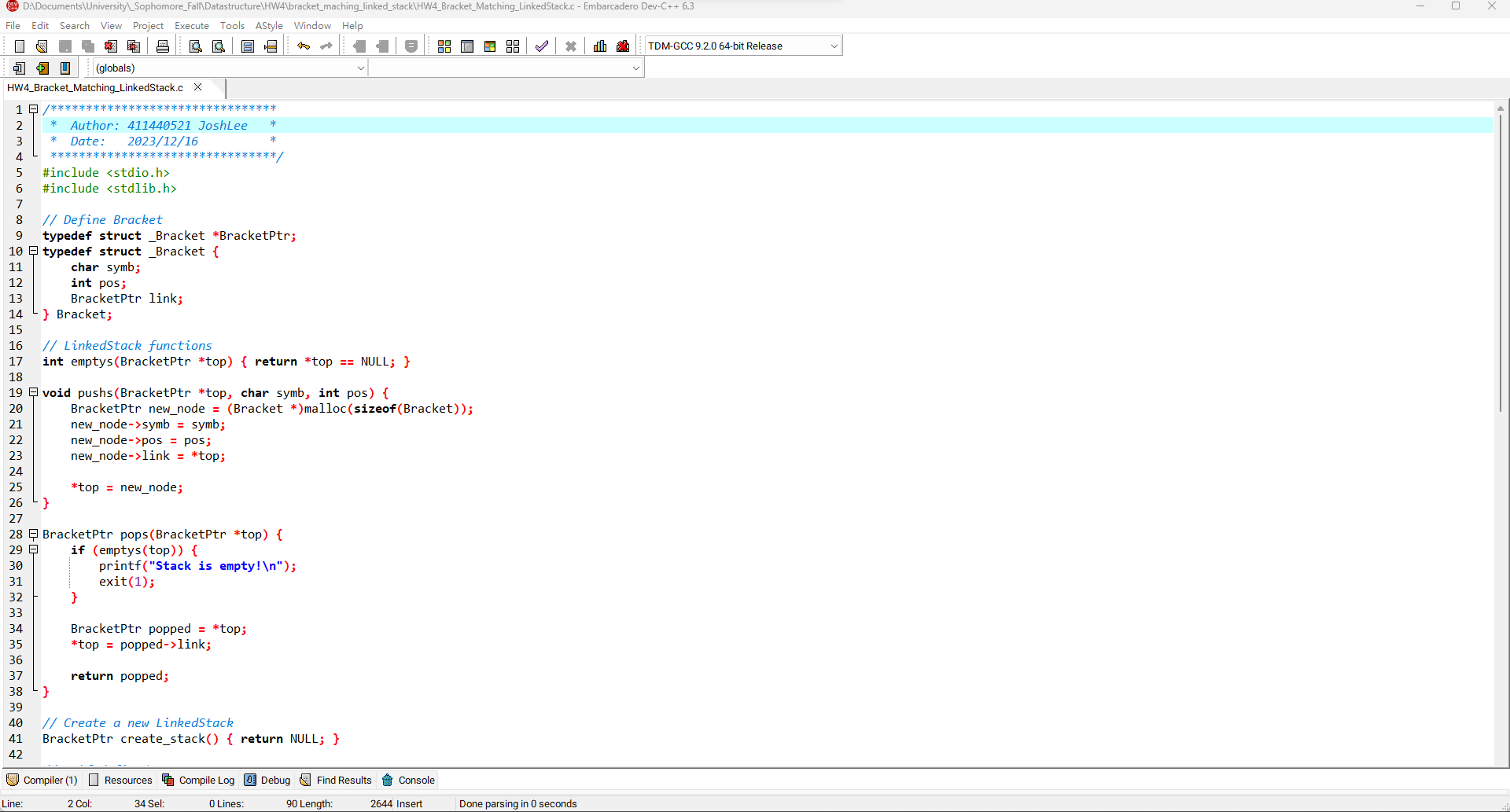
Pop()兩次  
([

7

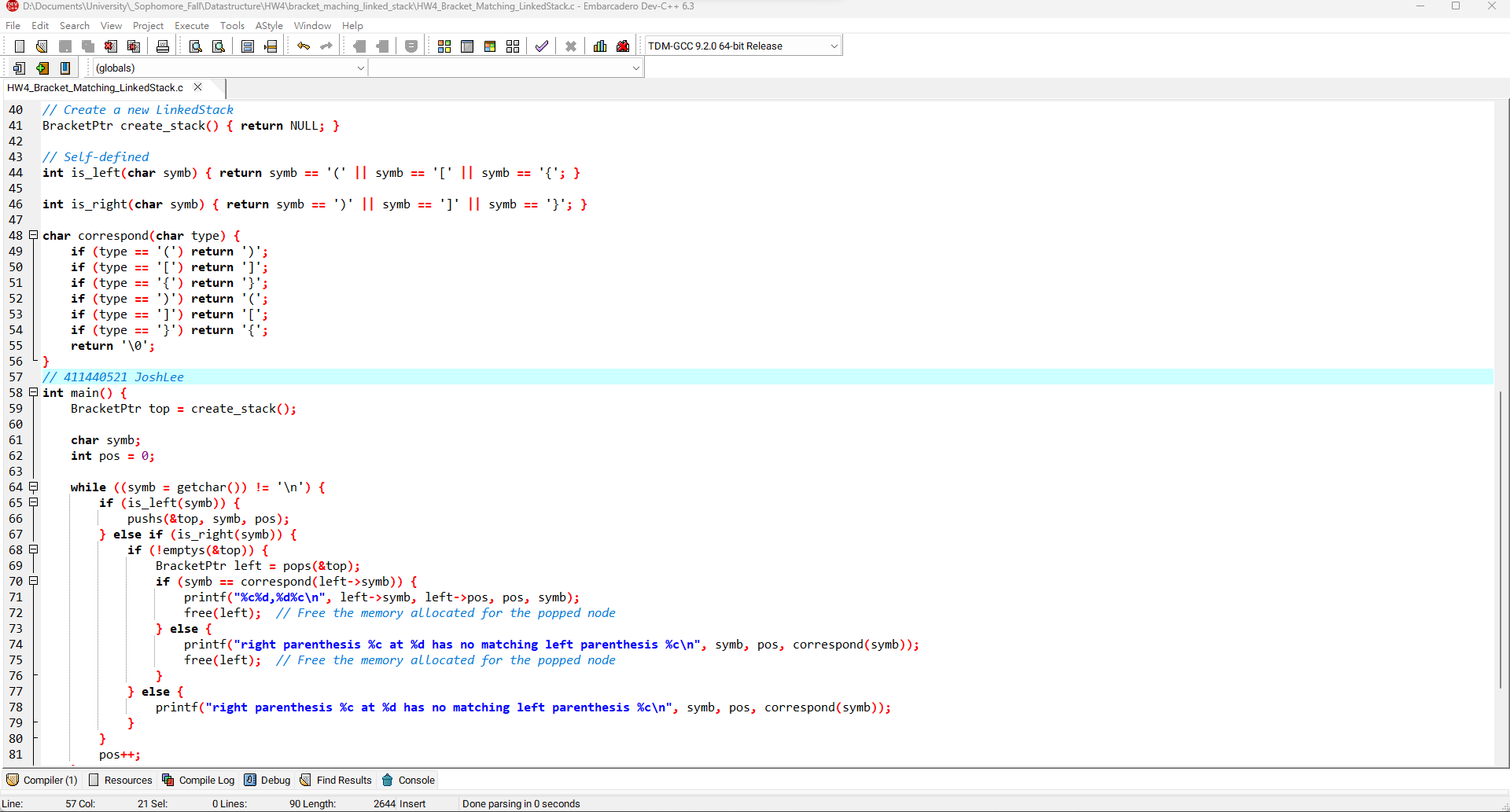
Pop()兩次結果  
([

Pop剩餘的

8

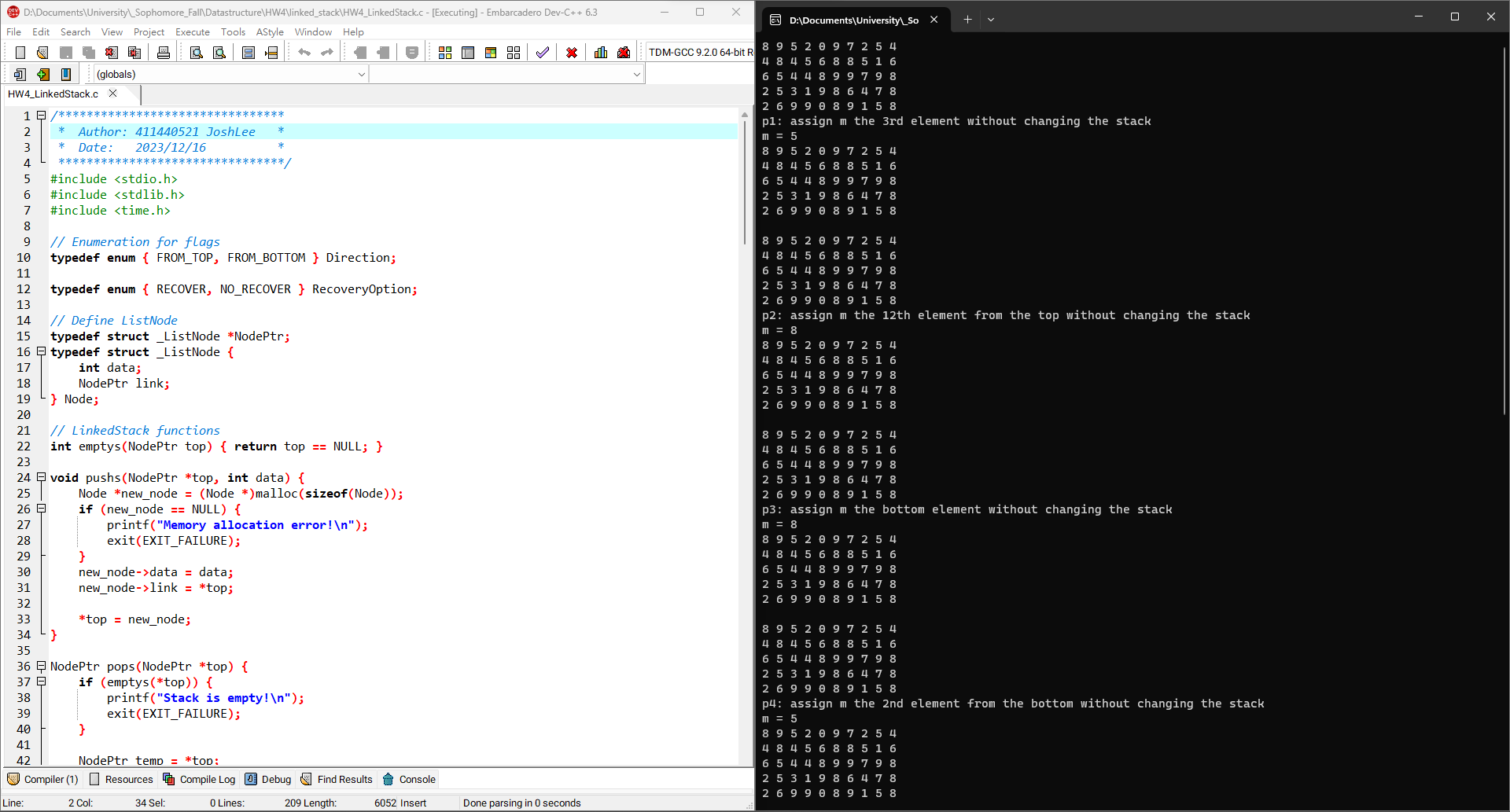
9

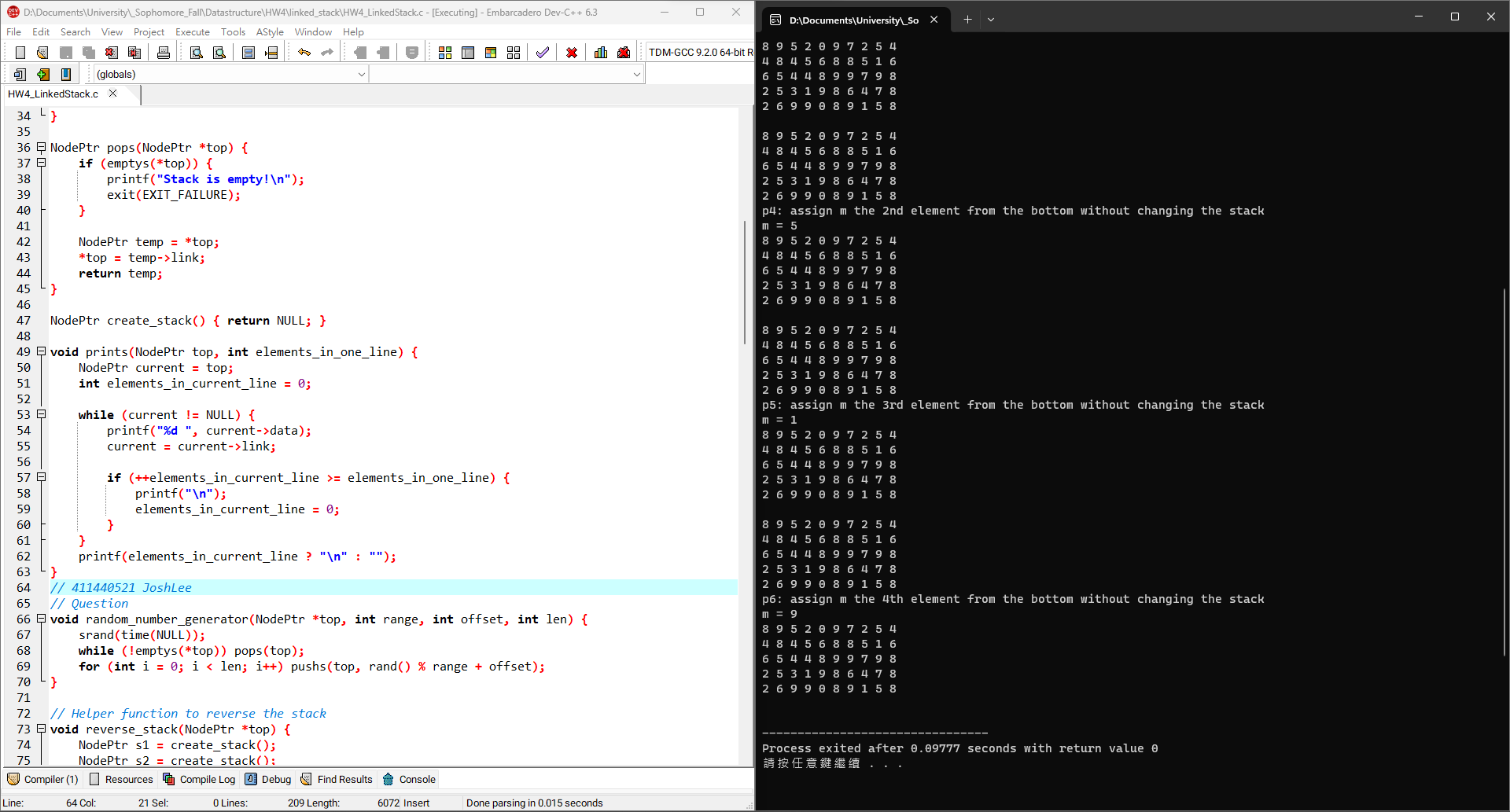
最終結果

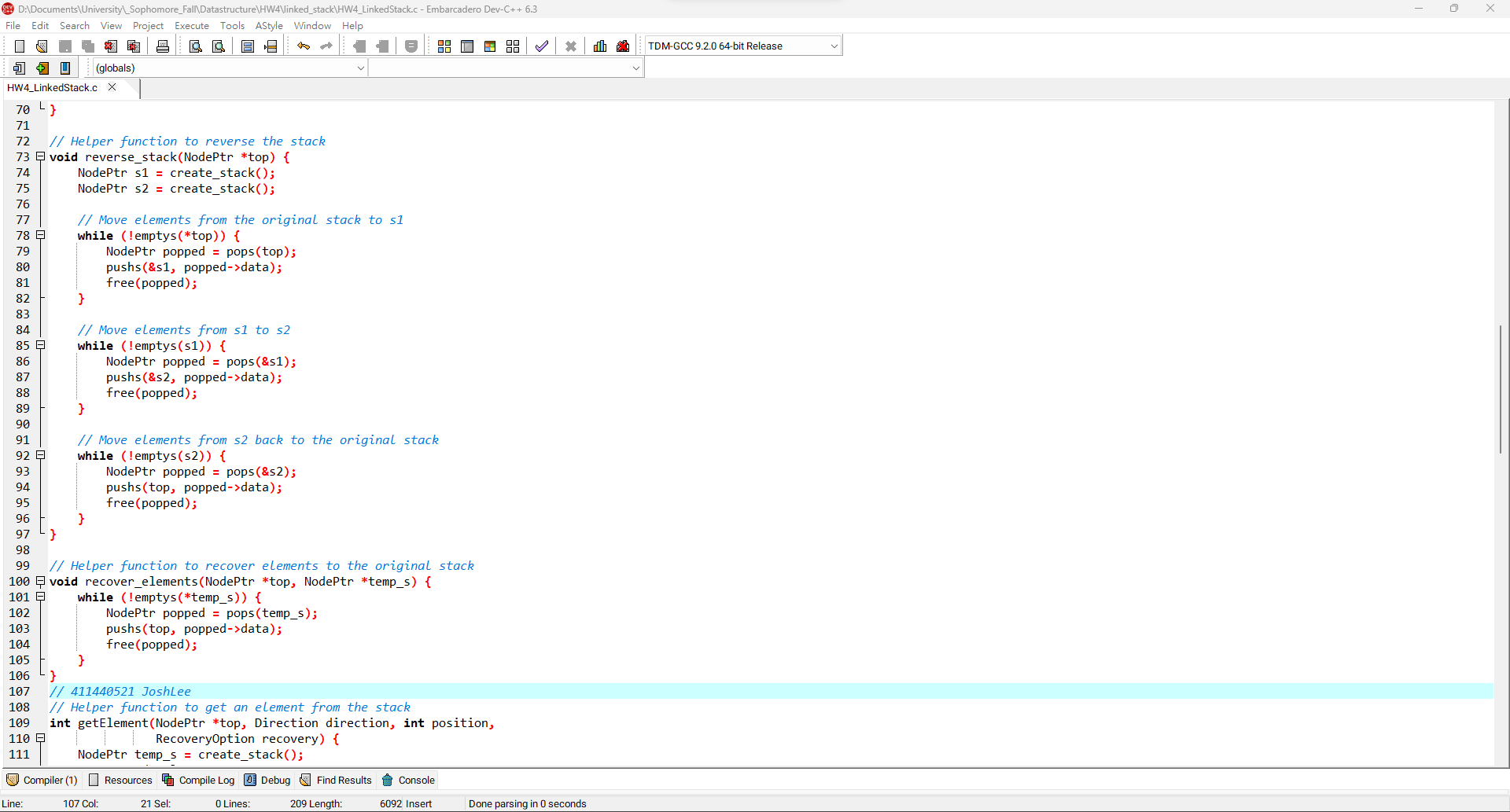
10

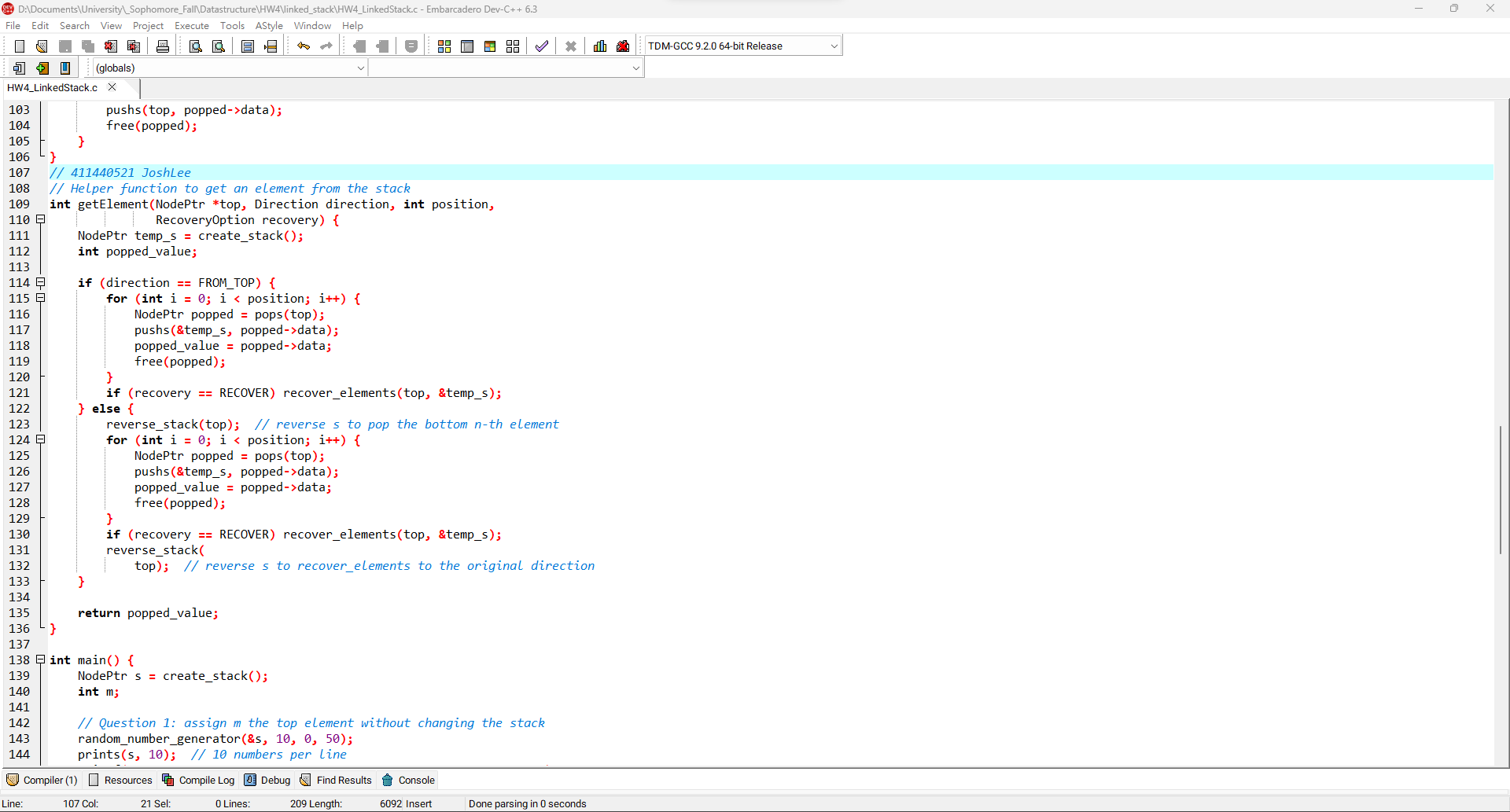
**Bracket Matching程式碼:**

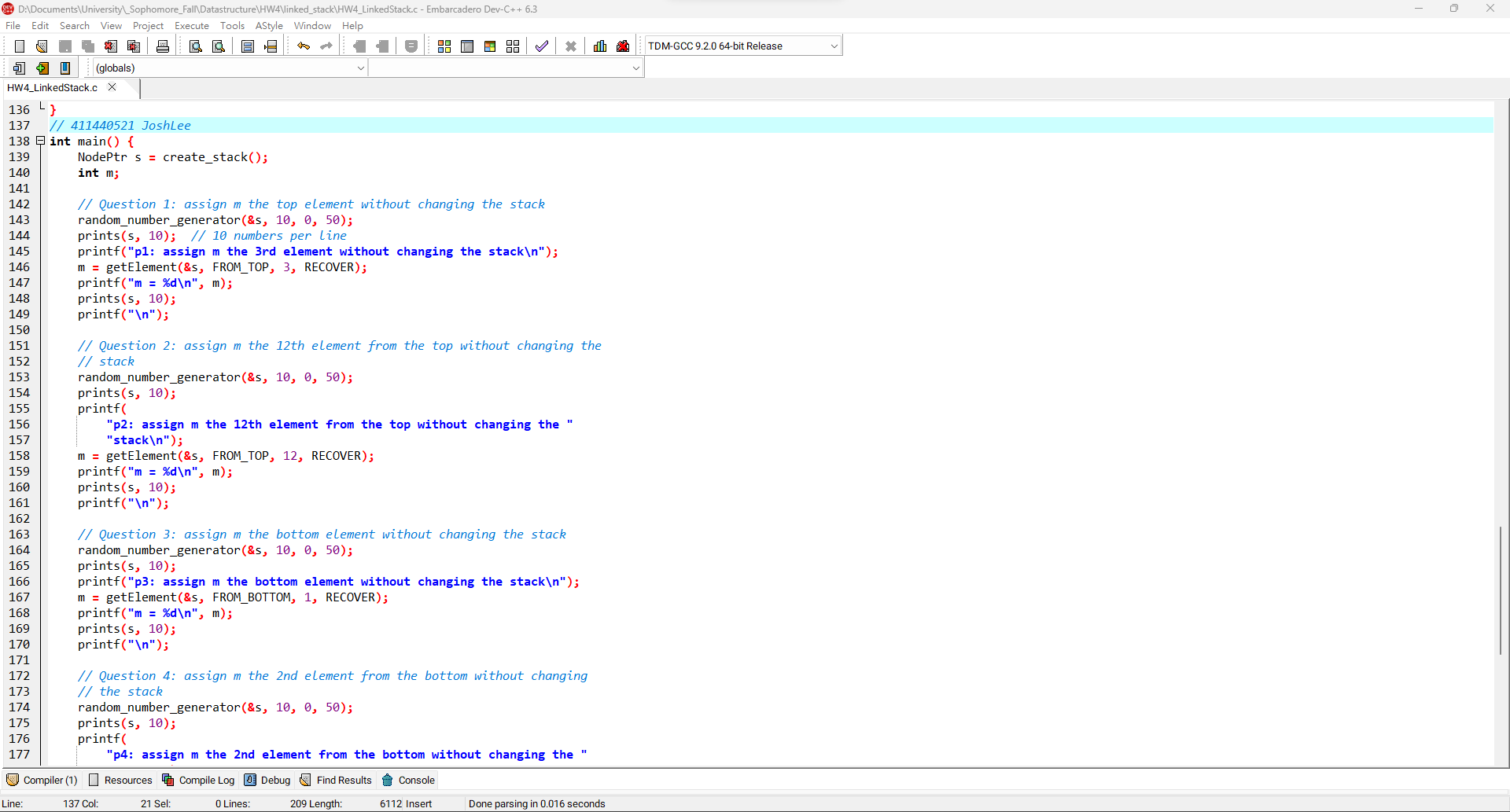
|  |  |
| --- | --- |
| #include <stdio.h>  #include <stdlib.h>  *// Define Bracket*  typedef struct \_Bracket \*BracketPtr;  typedef struct \_Bracket {  char symb;  int pos;  BracketPtr link;  } Bracket;  *// LinkedStack functions*  int emptys(BracketPtr \*top) { return \*top == NULL; }  void pushs(BracketPtr \*top, char symb, int pos) {  BracketPtr new\_node = (Bracket \*)malloc(sizeof(Bracket));  new\_node->symb = symb;  new\_node->pos = pos;  new\_node->link = \*top;  \*top = new\_node;  }  BracketPtr pops(BracketPtr \*top) {  if (emptys(top)) {  printf("Stack is empty!\n");  exit(1);  }  BracketPtr popped = \*top;  \*top = popped->link;  return popped;  }  *// Create a new LinkedStack*  BracketPtr create\_stack() { return NULL; }  *// Self-defined*  int is\_left(char symb) { return symb == '(' || symb == '[' || symb == '{'; }  int is\_right(char symb) { return symb == ')' || symb == ']' || symb == '}'; } | char correspond(char type) {  if (type == '(') return ')';  if (type == '[') return ']';  if (type == '{') return '}';  if (type == ')') return '(';  if (type == ']') return '[';  if (type == '}') return '{';  return '\0';  }  int main() {  BracketPtr top = create\_stack();  char symb;  int pos = 0;  while ((symb = getchar()) != '\n') {  if (is\_left(symb)) {  pushs(&top, symb, pos);  } else if (is\_right(symb)) {  if (!emptys(&top)) {  BracketPtr left = pops(&top);  if (symb == correspond(left->symb)) {  printf("%c%d,%d%c\n", left->symb, left->pos, pos, symb);  free(left); *// Free the memory allocated for the popped node*  } else {  printf("right parenthesis %c at %d has no matching left parenthesis %c\n", symb, pos, correspond(symb));  free(left); *// Free the memory allocated for the popped node*  }  } else {  printf("right parenthesis %c at %d has no matching left parenthesis %c\n", symb, pos, correspond(symb));  }  }  pos++;  }  while (!emptys(&top)) {  BracketPtr left = pops(&top);  printf("left parenthesis %c at %d has no matching right parenthesis %c\n", left->symb, left->pos, correspond(left->symb));  free(left); *// Free the memory allocated for the popped node*  }  return 0;  } |

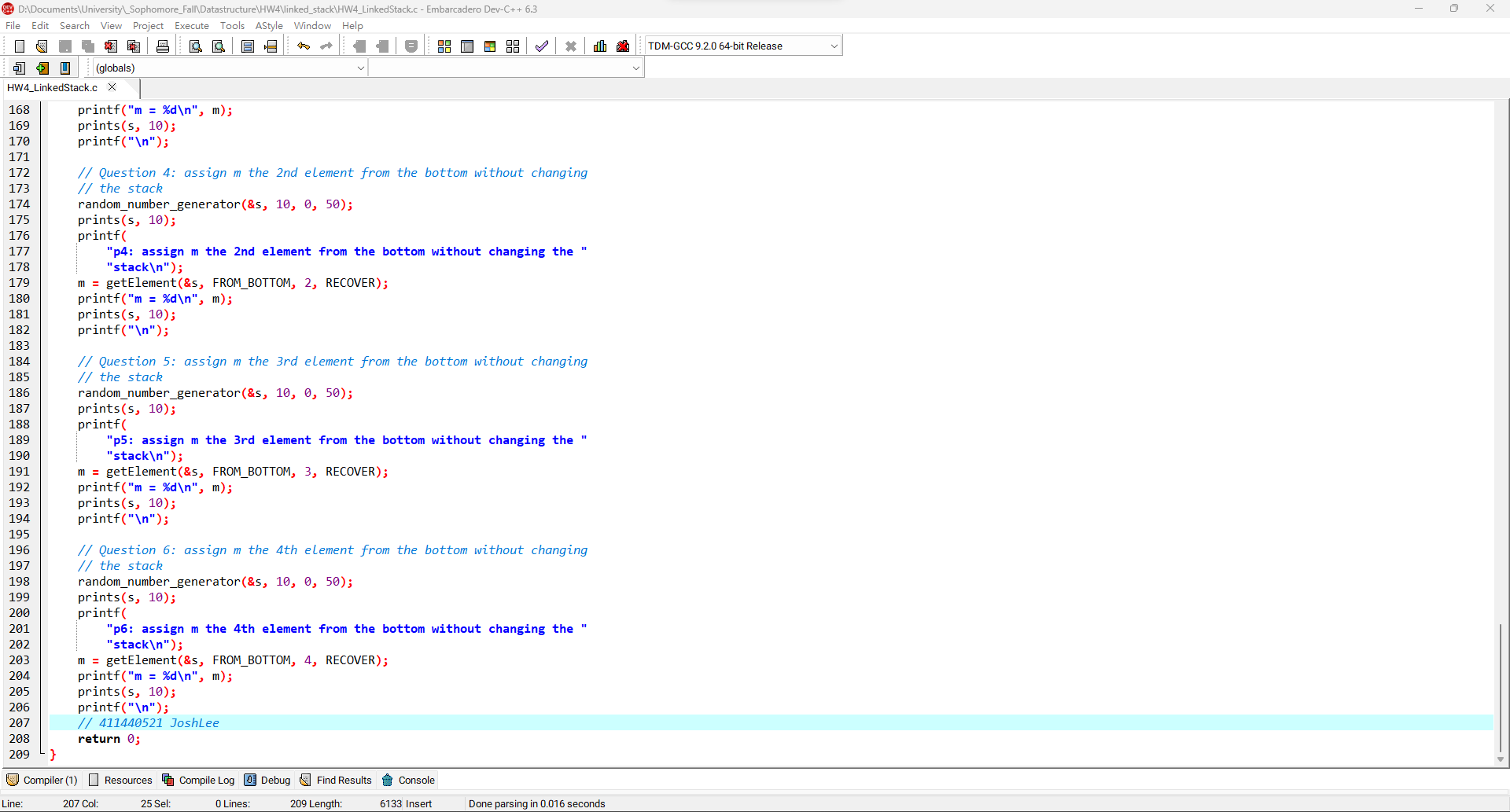
1

2

3

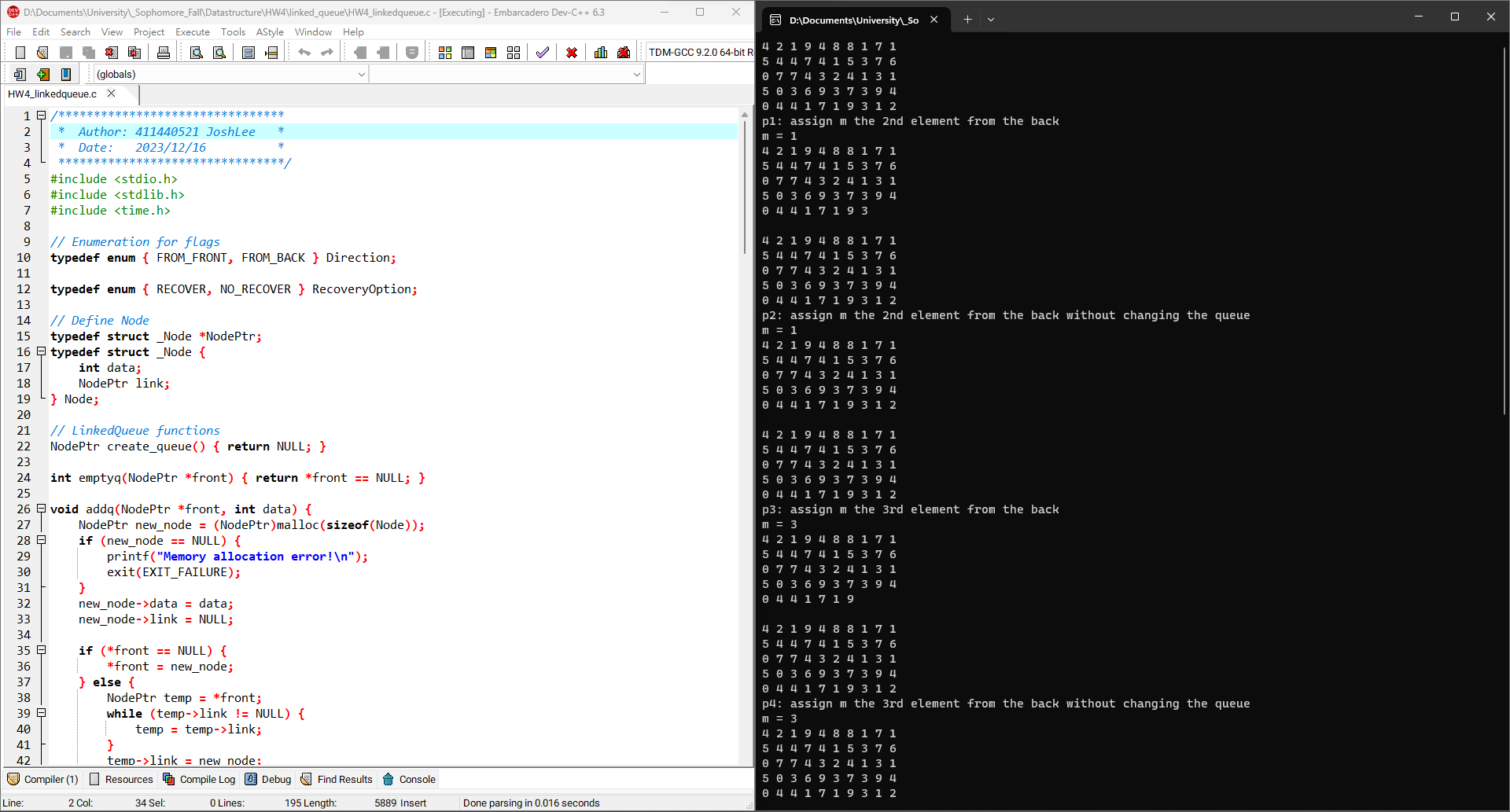
4

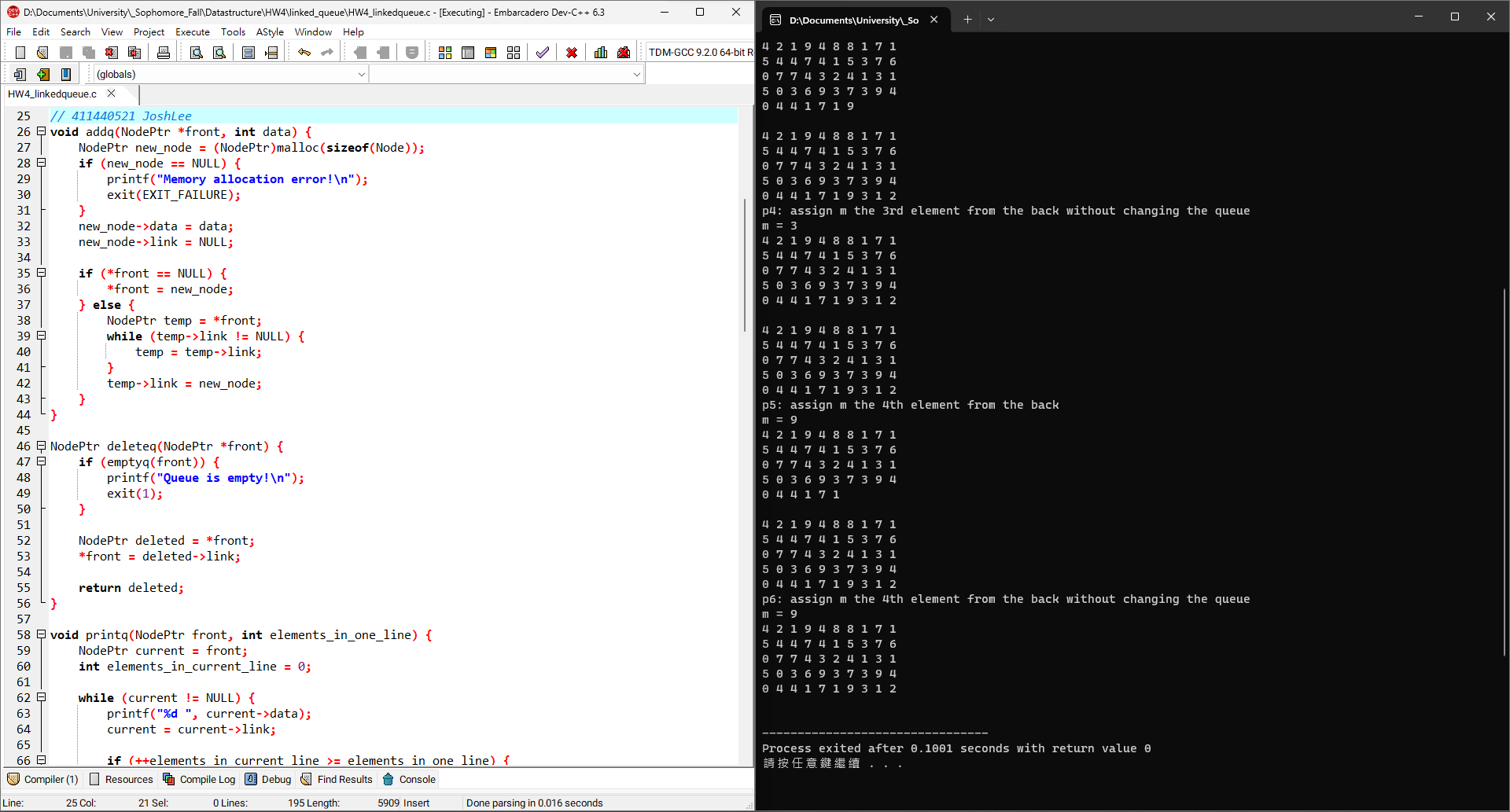
5

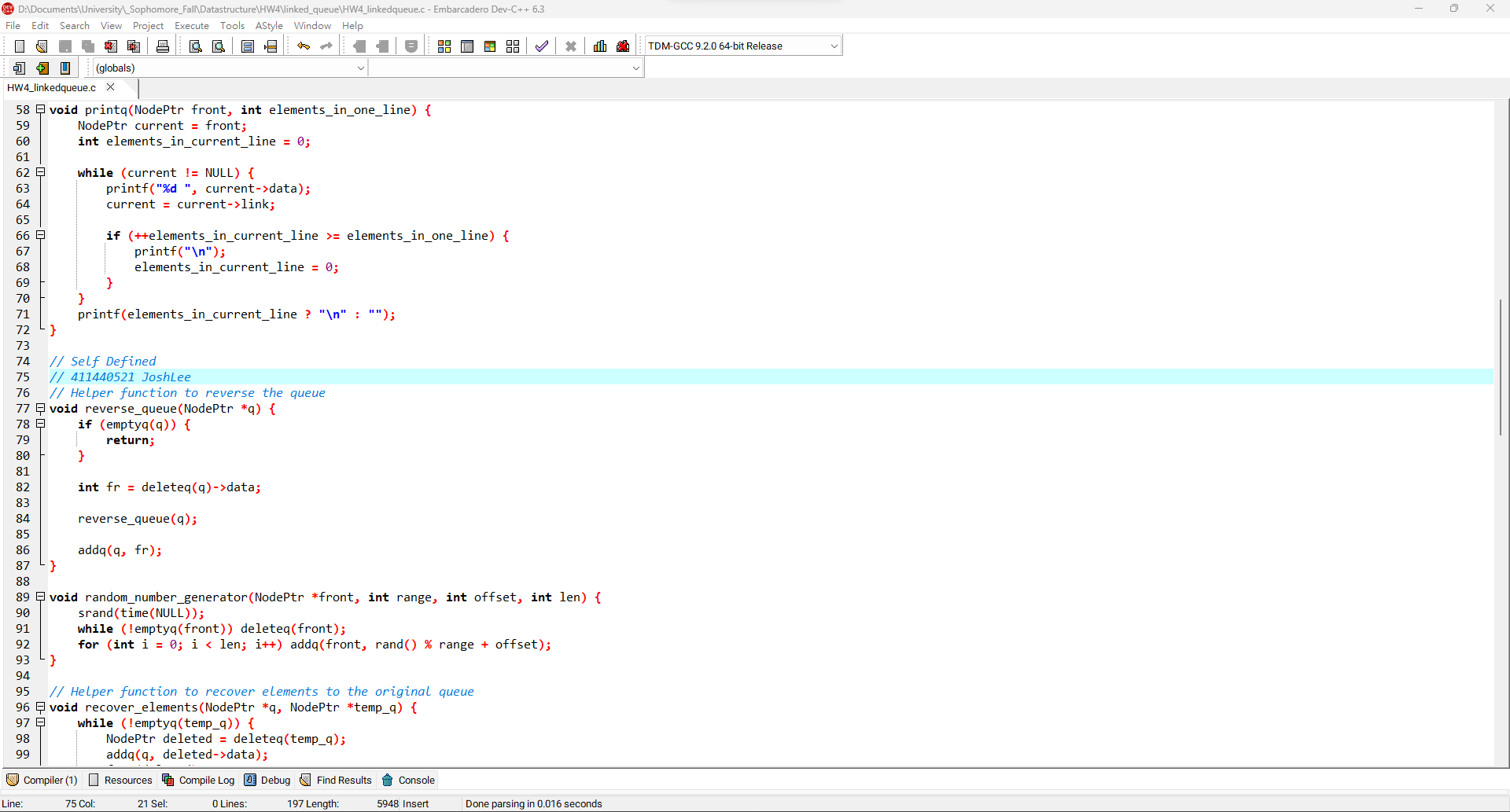
6

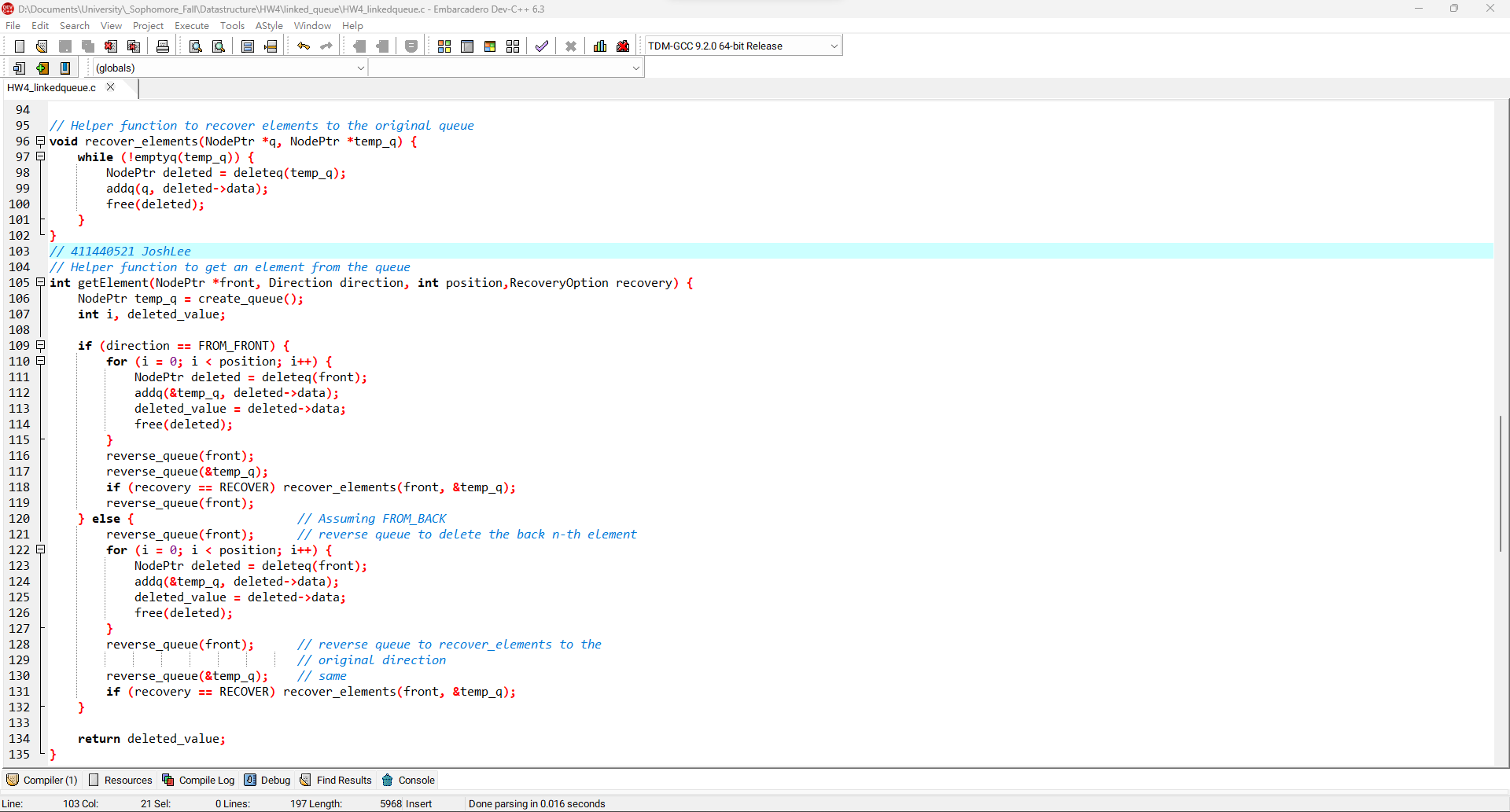
**LinkedStack程式碼**

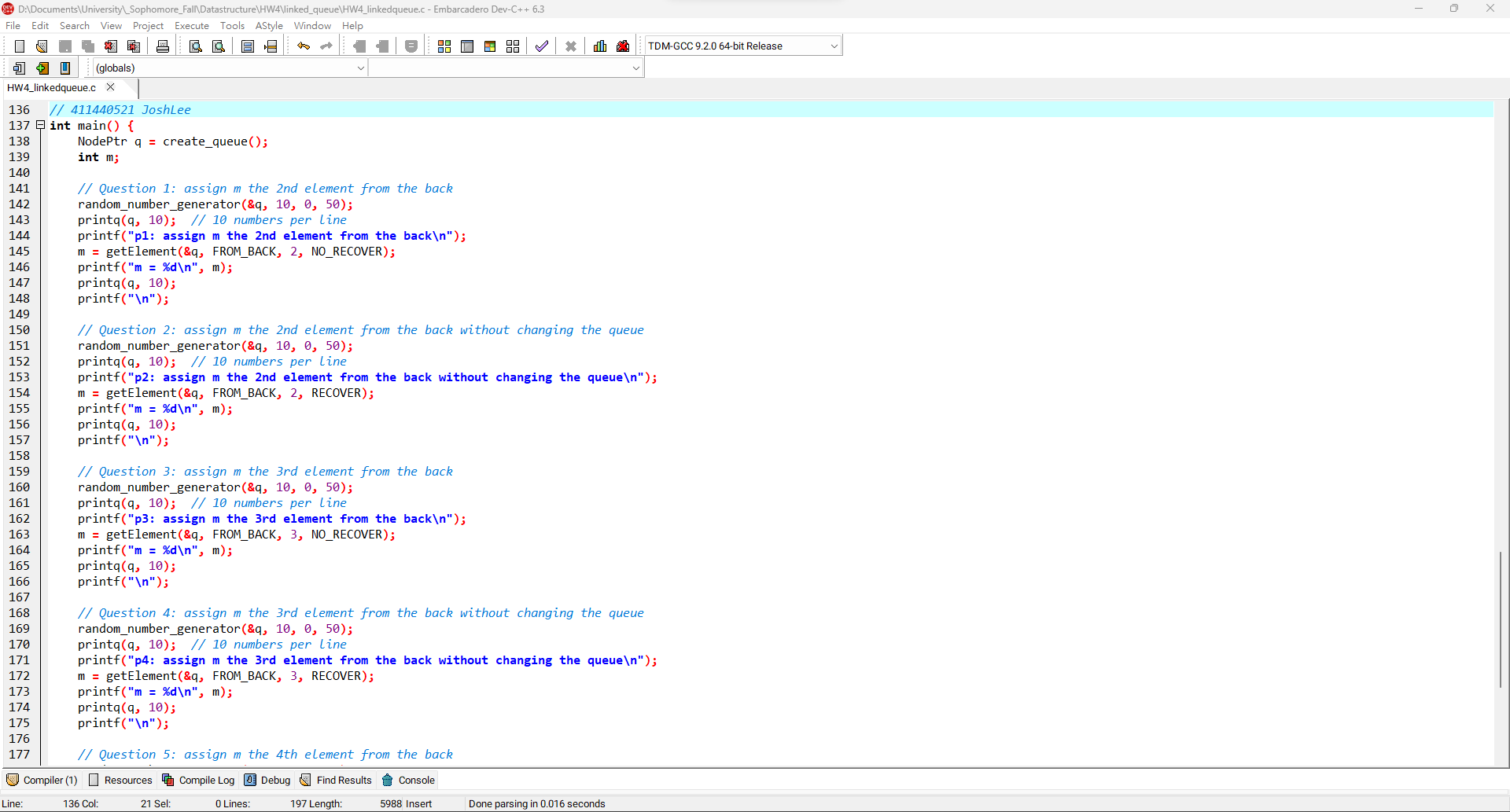
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  *// Enumeration for flags*  typedef enum { FROM\_TOP, FROM\_BOTTOM } Direction;  typedef enum { RECOVER, NO\_RECOVER } RecoveryOption;  *// Define Node*  typedef struct \_Node \*NodePtr;  typedef struct \_Node {  int data;  NodePtr link;  } Node;  *// LinkedStack functions*  int emptys(NodePtr \*top) { return \*top == NULL; }  void pushs(NodePtr \*top, int data) {  Node \*new\_node = (Node \*)malloc(sizeof(Node));  if (new\_node == NULL) {  printf("Memory allocation error!\n");  exit(EXIT\_FAILURE);  }  new\_node->data = data;  new\_node->link = \*top;  \*top = new\_node;  }  NodePtr pops(NodePtr \*top) {  if (emptys(top)) {  printf("Stack is empty!\n");  exit(EXIT\_FAILURE);  }  NodePtr popped = \*top;  \*top = popped->link;  return popped;  }  NodePtr create\_stack() { return NULL; }  void prints(NodePtr top, int elements\_in\_one\_line) {  NodePtr current = top;  int elements\_in\_current\_line = 0;  while (current != NULL) {  printf("%d ", current->data);  current = current->link;  if (++elements\_in\_current\_line >= elements\_in\_one\_line) {  printf("\n");  elements\_in\_current\_line = 0;  }  }  printf(elements\_in\_current\_line ? "\n" : "");  }  *// Question*  void random\_number\_generator(NodePtr \*top, int range, int offset, int len) {  srand(time(NULL));  while (!emptys(top)) pops(top);  for (int i = 0; i < len; i++) pushs(top, rand() % range + offset);  }  *// Helper function to reverse the stack*  void reverse\_stack(NodePtr \*top) {  NodePtr s1 = create\_stack();  NodePtr s2 = create\_stack();  *// Move elements from the original stack to s1*  while (!emptys(top)) {  NodePtr popped = pops(top);  pushs(&s1, popped->data);  free(popped);  }  *// Move elements from s1 to s2*  while (!emptys(&s1)) {  NodePtr popped = pops(&s1);  pushs(&s2, popped->data);  free(popped);  }  *// Move elements from s2 back to the original stack*  while (!emptys(&s2)) {  NodePtr popped = pops(&s2);  pushs(top, popped->data);  free(popped);  }  }  *// Helper function to recover elements to the original stack*  void recover\_elements(NodePtr \*top, NodePtr \*temp\_s) {  while (!emptys(temp\_s)) {  NodePtr popped = pops(temp\_s);  pushs(top, popped->data);  free(popped);  }  }  *// Helper function to get an element from the stack*  int getElement(NodePtr \*top, Direction direction, int position, RecoveryOption recovery) {  NodePtr temp\_s = create\_stack();  int popped\_value;  if (direction == FROM\_TOP) {  for (int i = 0; i < position; i++) {  NodePtr popped = pops(top);  pushs(&temp\_s, popped->data);  popped\_value = popped->data;  free(popped);  }  if (recovery == RECOVER) recover\_elements(top, &temp\_s);  } else {  reverse\_stack(top); *// reverse s to pop the bottom n-th element*  for (int i = 0; i < position; i++) {  NodePtr popped = pops(top);  pushs(&temp\_s, popped->data);  popped\_value = popped->data;  free(popped);  }  if (recovery == RECOVER) recover\_elements(top, &temp\_s);  reverse\_stack(top); *// reverse s to recover\_elements to the original direction*  }  return popped\_value;  }  int main() {  NodePtr s = create\_stack();  int m;  *// Question 1: assign m the top element without changing the stack*  random\_number\_generator(&s, 10, 0, 50);  prints(s, 10); *// 10 numbers per line*  printf("p1: assign m the 3rd element from the top without changing the stack\n");  m = getElement(&s, FROM\_TOP, 3, RECOVER);  printf("m = %d\n", m);  prints(s, 10);  printf("\n");  *// Question 2: assign m the 12th element from the top without changing the stack*  random\_number\_generator(&s, 10, 0, 50);  prints(s, 10);  printf("p2: assign m the 12th element from the top without changing the stack\n");  m = getElement(&s, FROM\_TOP, 12, RECOVER);  printf("m = %d\n", m);  prints(s, 10);  printf("\n");  *// Question 3: assign m the bottom element without changing the stack*  random\_number\_generator(&s, 10, 0, 50);  prints(s, 10);  printf("p3: assign m the bottom element without changing the stack\n");  m = getElement(&s, FROM\_BOTTOM, 1, RECOVER);  printf("m = %d\n", m);  prints(s, 10);  printf("\n");  *// Question 4: assign m the 2nd element from the bottom without changing the stack*  random\_number\_generator(&s, 10, 0, 50);  prints(s, 10);  printf("p4: assign m the 2nd element from the bottom without changing the stack\n");  m = getElement(&s, FROM\_BOTTOM, 2, RECOVER);  printf("m = %d\n", m);  prints(s, 10);  printf("\n");  *// Question 5: assign m the 3rd element from the bottom without changing the stack*  random\_number\_generator(&s, 10, 0, 50);  prints(s, 10);  printf("p5: assign m the 3rd element from the bottom without changing the stack\n");  m = getElement(&s, FROM\_BOTTOM, 3, RECOVER);  printf("m = %d\n", m);  prints(s, 10);  printf("\n");  *// Question 6: assign m the 4th element from the bottom without changing the stack*  random\_number\_generator(&s, 10, 0, 50);  prints(s, 10);  printf("p6: assign m the 4th element from the bottom without changing the stack\n");  m = getElement(&s, FROM\_BOTTOM, 4, RECOVER);  printf("m = %d\n", m);  prints(s, 10);  printf("\n");  return 0;  } |

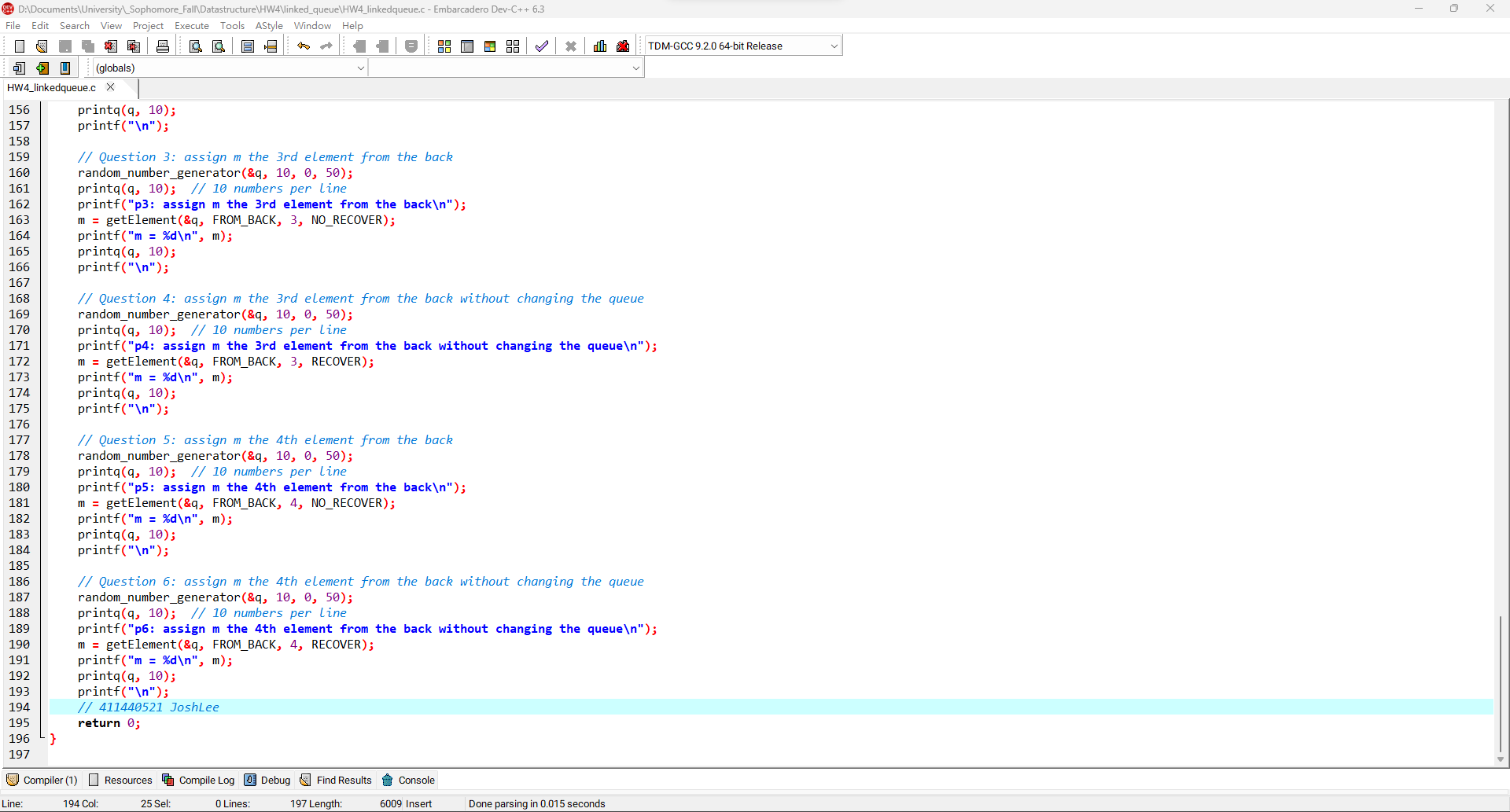
1

2

3

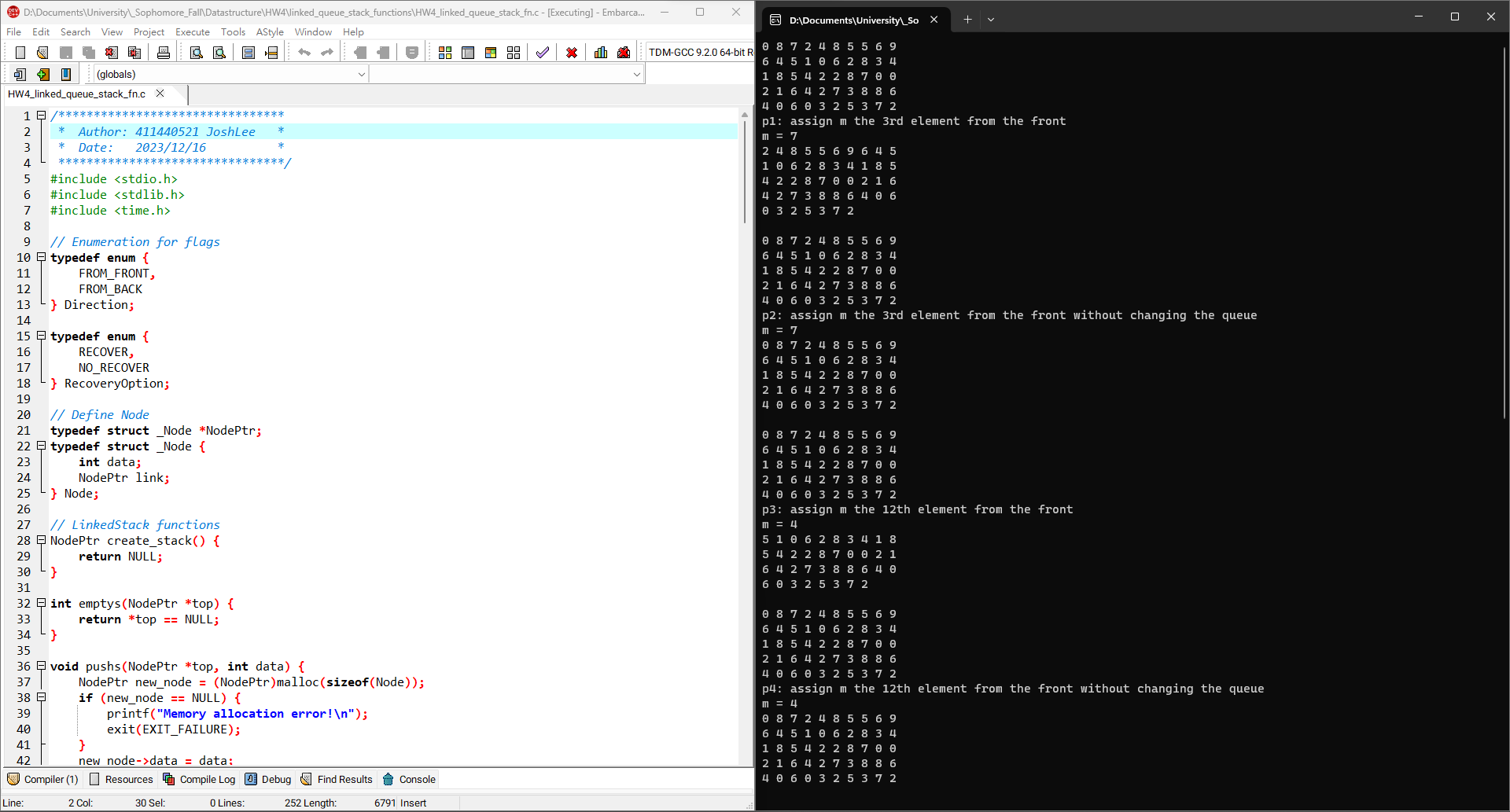
4

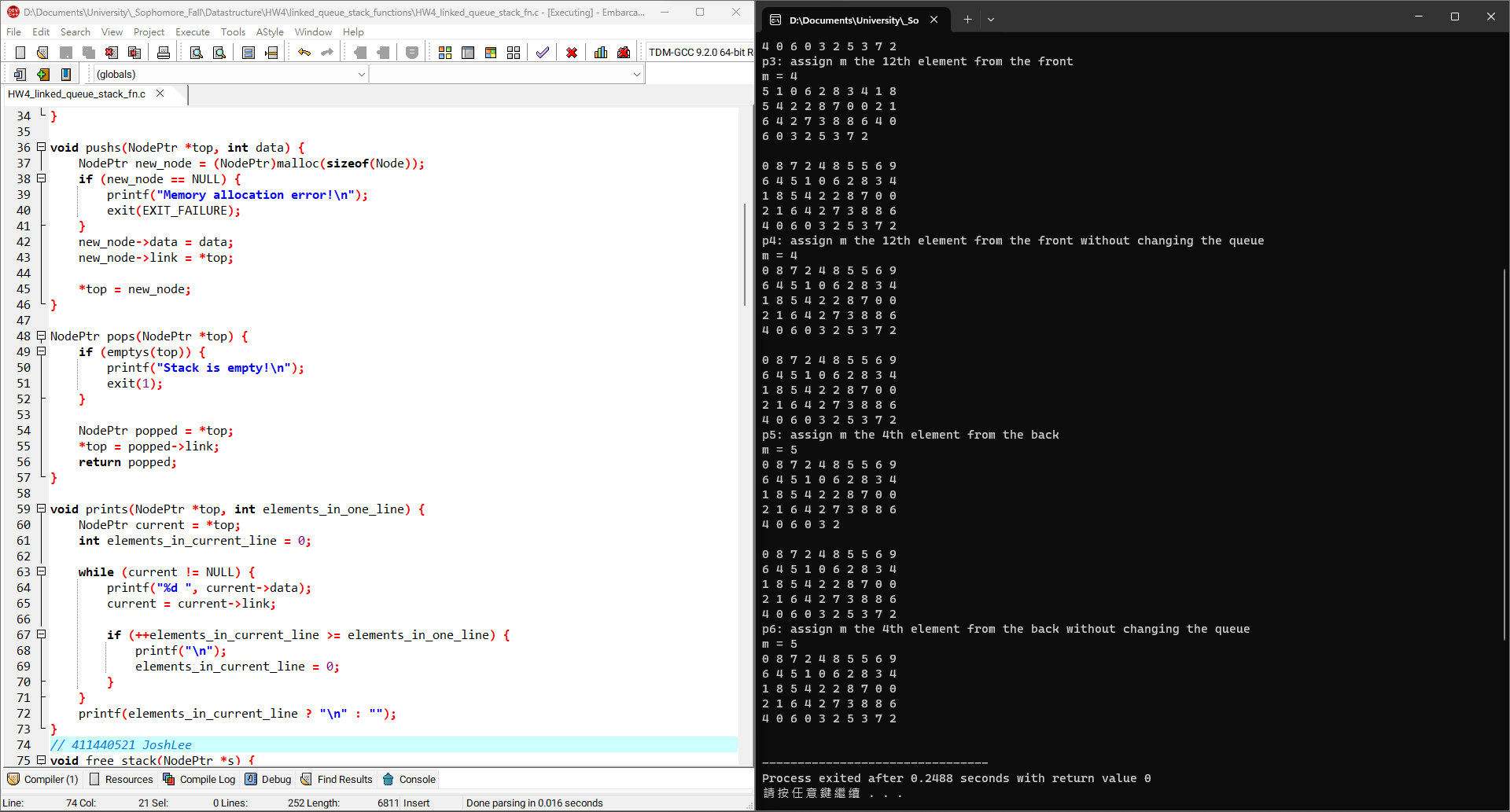
5

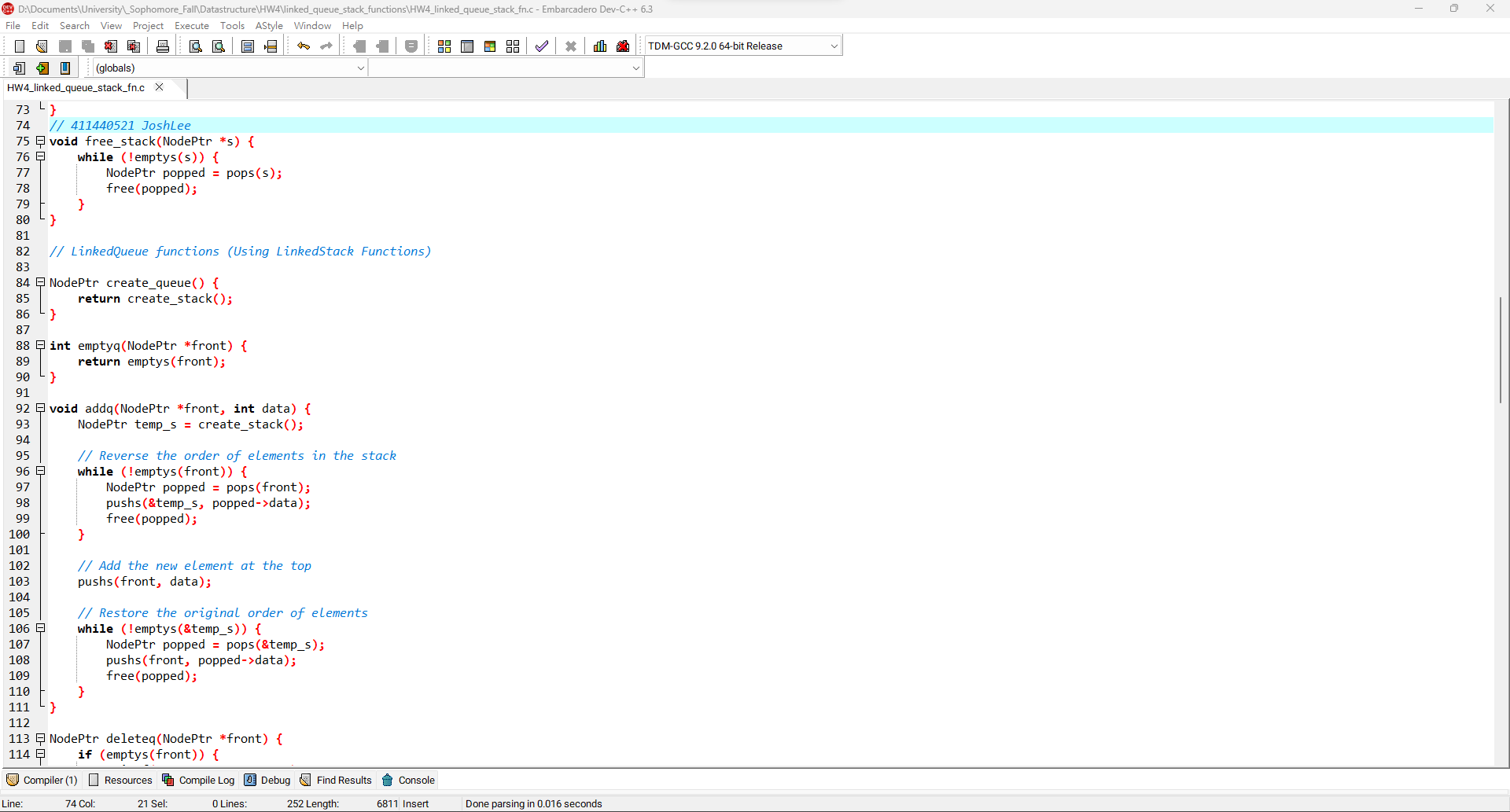
6

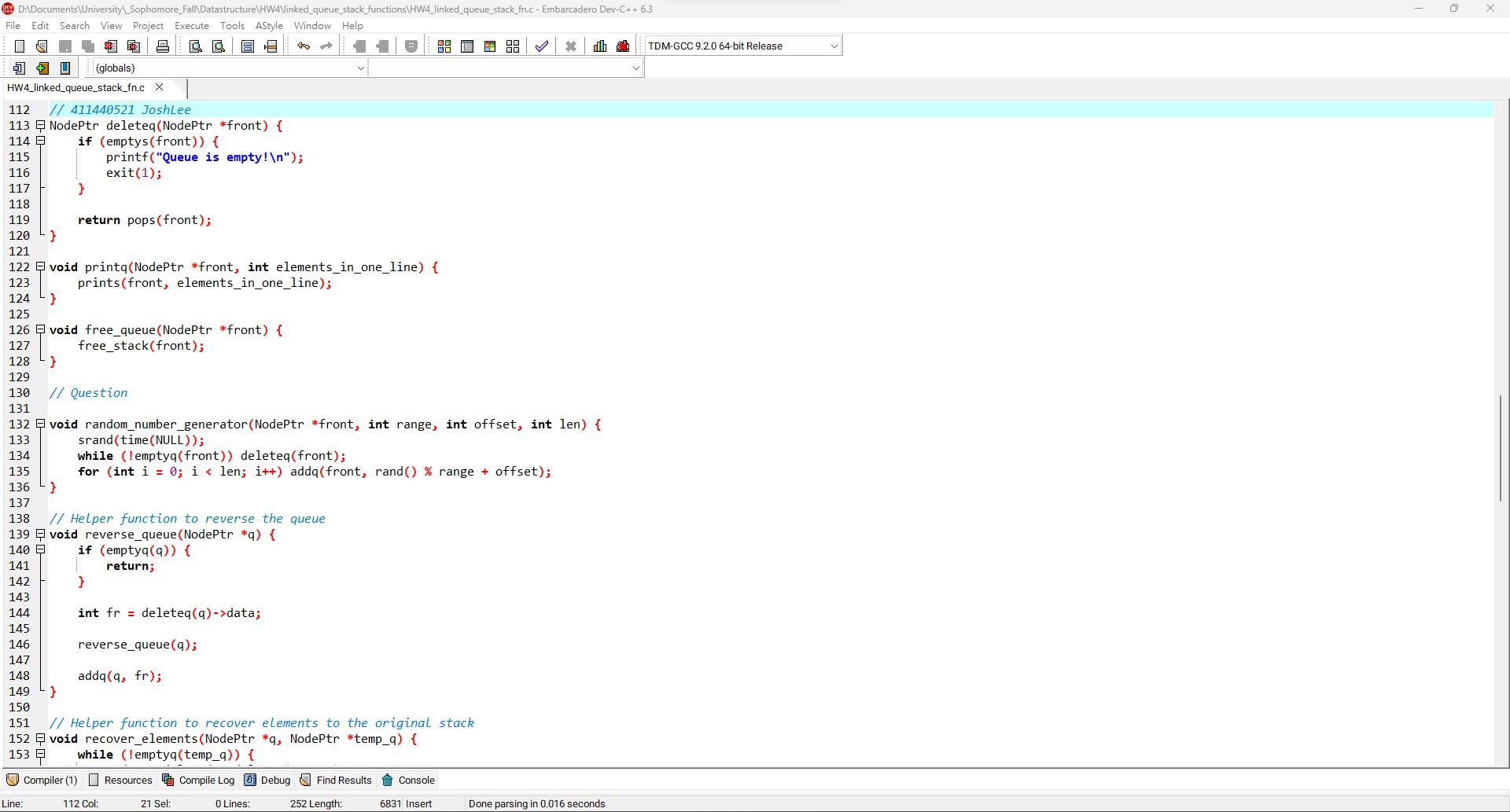
**LinkedQueue程式碼:**

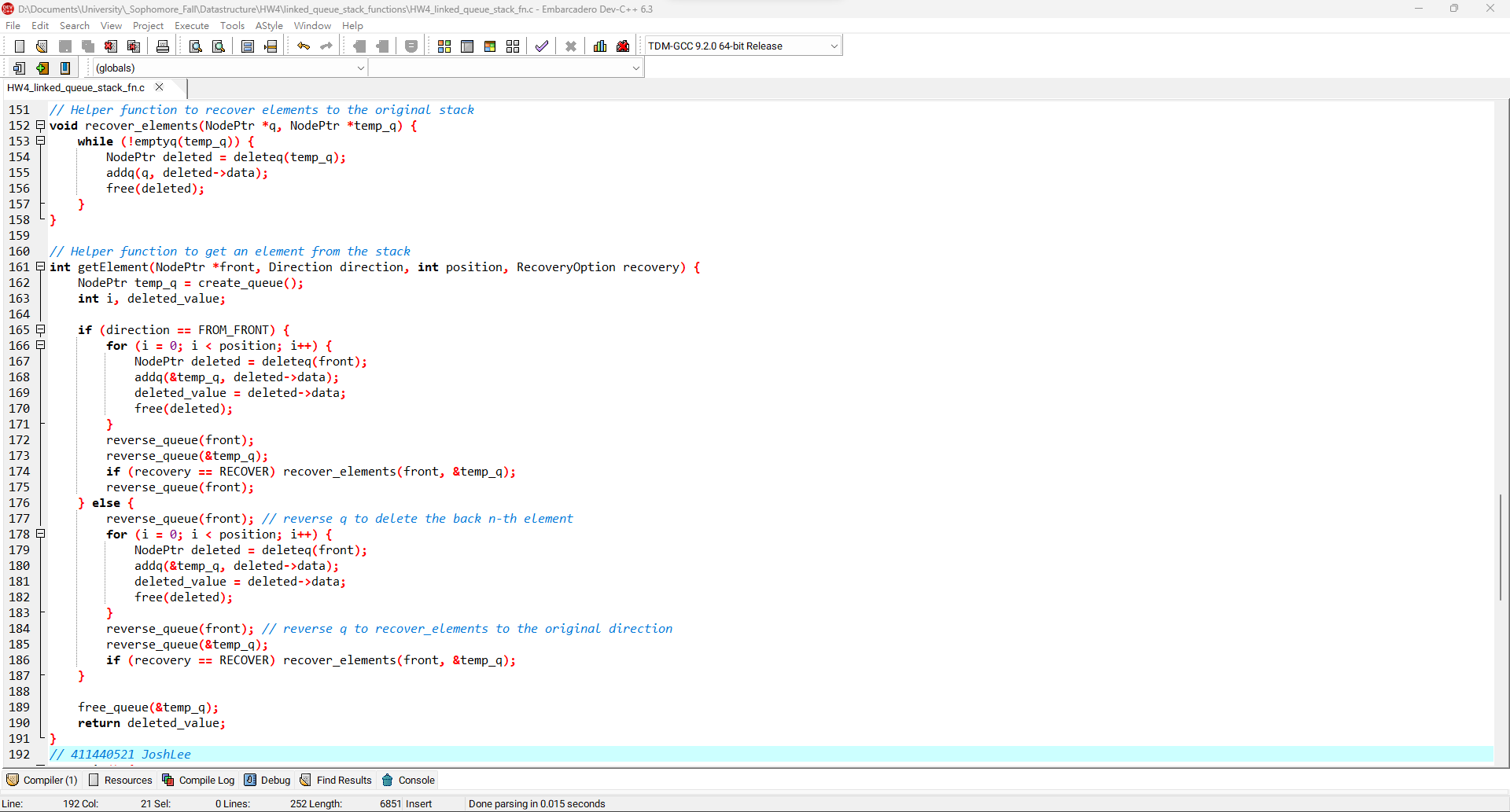
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  *// Enumeration for flags*  typedef enum { FROM\_FRONT, FROM\_BACK } Direction;  typedef enum { RECOVER, NO\_RECOVER } RecoveryOption;  *// Define Node*  typedef struct \_Node \*NodePtr;  typedef struct \_Node {  int data;  NodePtr link;  } Node;  *// LinkedQueue functions*  NodePtr create\_queue() { return NULL; }  int emptyq(NodePtr \*front) { return \*front == NULL; }  void addq(NodePtr \*front, int data) {  NodePtr new\_node = (NodePtr)malloc(sizeof(Node));  if (new\_node == NULL) {  printf("Memory allocation error!\n");  exit(EXIT\_FAILURE);  }  new\_node->data = data;  new\_node->link = NULL;  if (\*front == NULL) {  \*front = new\_node;  } else {  NodePtr temp = \*front;  while (temp->link != NULL) {  temp = temp->link;  }  temp->link = new\_node;  }  }  NodePtr deleteq(NodePtr \*front) {  if (emptyq(front)) {  printf("Queue is empty!\n");  exit(1);  }  NodePtr deleted = \*front;  \*front = deleted->link;  return deleted;  }  void printq(NodePtr front, int elements\_in\_one\_line) {  NodePtr current = front;  int elements\_in\_current\_line = 0;  while (current != NULL) {  printf("%d ", current->data);  current = current->link;  if (++elements\_in\_current\_line >= elements\_in\_one\_line) {  printf("\n");  elements\_in\_current\_line = 0;  }  }  printf(elements\_in\_current\_line ? "\n" : "");  }  *// Helper function to reverse the queue*  void reverse\_queue(NodePtr \*q) {  if (emptyq(q)) {  return;  }  int fr = deleteq(q)->data;  reverse\_queue(q);  addq(q, fr);  }  void random\_number\_generator(NodePtr \*front, int range, int offset, int len) {  srand(time(NULL));  while (!emptyq(front)) deleteq(front);  for (int i = 0; i < len; i++) addq(front, rand() % range + offset);  }  *// Helper function to recover elements to the original queue*  void recover\_elements(NodePtr \*q, NodePtr \*temp\_q) {  while (!emptyq(temp\_q)) {  NodePtr deleted = deleteq(temp\_q);  addq(q, deleted->data);  free(deleted);  }  }  *// Helper function to get an element from the queue*  int getElement(NodePtr \*front, Direction direction, int position,RecoveryOption recovery) {  NodePtr temp\_q = create\_queue();  int i, deleted\_value;  if (direction == FROM\_FRONT) {  for (i = 0; i < position; i++) {  NodePtr deleted = deleteq(front);  addq(&temp\_q, deleted->data);  deleted\_value = deleted->data;  free(deleted);  }  reverse\_queue(front);  reverse\_queue(&temp\_q);  if (recovery == RECOVER) recover\_elements(front, &temp\_q);  reverse\_queue(front);  } else { *// Assuming FROM\_BACK*  reverse\_queue(front); *// reverse queue to delete the back n-th element*  for (i = 0; i < position; i++) {  NodePtr deleted = deleteq(front);  addq(&temp\_q, deleted->data);  deleted\_value = deleted->data;  free(deleted);  }  reverse\_queue(front); *// reverse queue to recover\_elements to the*  *// original direction*  reverse\_queue(&temp\_q); *// same*  if (recovery == RECOVER) recover\_elements(front, &temp\_q);  }  return deleted\_value;  }  int main() {  NodePtr q = create\_queue();  int m;  *// Question 1: assign m the 2nd element from the back*  random\_number\_generator(&q, 10, 0, 50);  printq(q, 10); *// 10 numbers per line*  printf("p1: assign m the 2nd element from the back\n");  m = getElement(&q, FROM\_BACK, 2, NO\_RECOVER);  printf("m = %d\n", m);  printq(q, 10);  printf("\n");  *// Question 2: assign m the 2nd element from the back without changing the queue*  random\_number\_generator(&q, 10, 0, 50);  printq(q, 10); *// 10 numbers per line*  printf("p2: assign m the 2nd element from the back without changing the queue\n");  m = getElement(&q, FROM\_BACK, 2, RECOVER);  printf("m = %d\n", m);  printq(q, 10);  printf("\n");  *// Question 3: assign m the 3rd element from the back*  random\_number\_generator(&q, 10, 0, 50);  printq(q, 10); *// 10 numbers per line*  printf("p3: assign m the 3rd element from the back\n");  m = getElement(&q, FROM\_BACK, 3, NO\_RECOVER);  printf("m = %d\n", m);  printq(q, 10);  printf("\n");  *// Question 4: assign m the 3rd element from the back without changing the queue*  random\_number\_generator(&q, 10, 0, 50);  printq(q, 10); *// 10 numbers per line*  printf("p4: assign m the 3rd element from the back without changing the queue\n");  m = getElement(&q, FROM\_BACK, 3, RECOVER);  printf("m = %d\n", m);  printq(q, 10);  printf("\n");  *// Question 5: assign m the 4th element from the back*  random\_number\_generator(&q, 10, 0, 50);  printq(q, 10); *// 10 numbers per line*  printf("p5: assign m the 4th element from the back\n");  m = getElement(&q, FROM\_BACK, 4, NO\_RECOVER);  printf("m = %d\n", m);  printq(q, 10);  printf("\n");  *// Question 6: assign m the 4th element from the back without changing the queue*  random\_number\_generator(&q, 10, 0, 50);  printq(q, 10); *// 10 numbers per line*  printf("p6: assign m the 4th element from the back without changing the queue\n");  m = getElement(&q, FROM\_BACK, 4, RECOVER);  printf("m = %d\n", m);  printq(q, 10);  printf("\n");  return 0;  } |

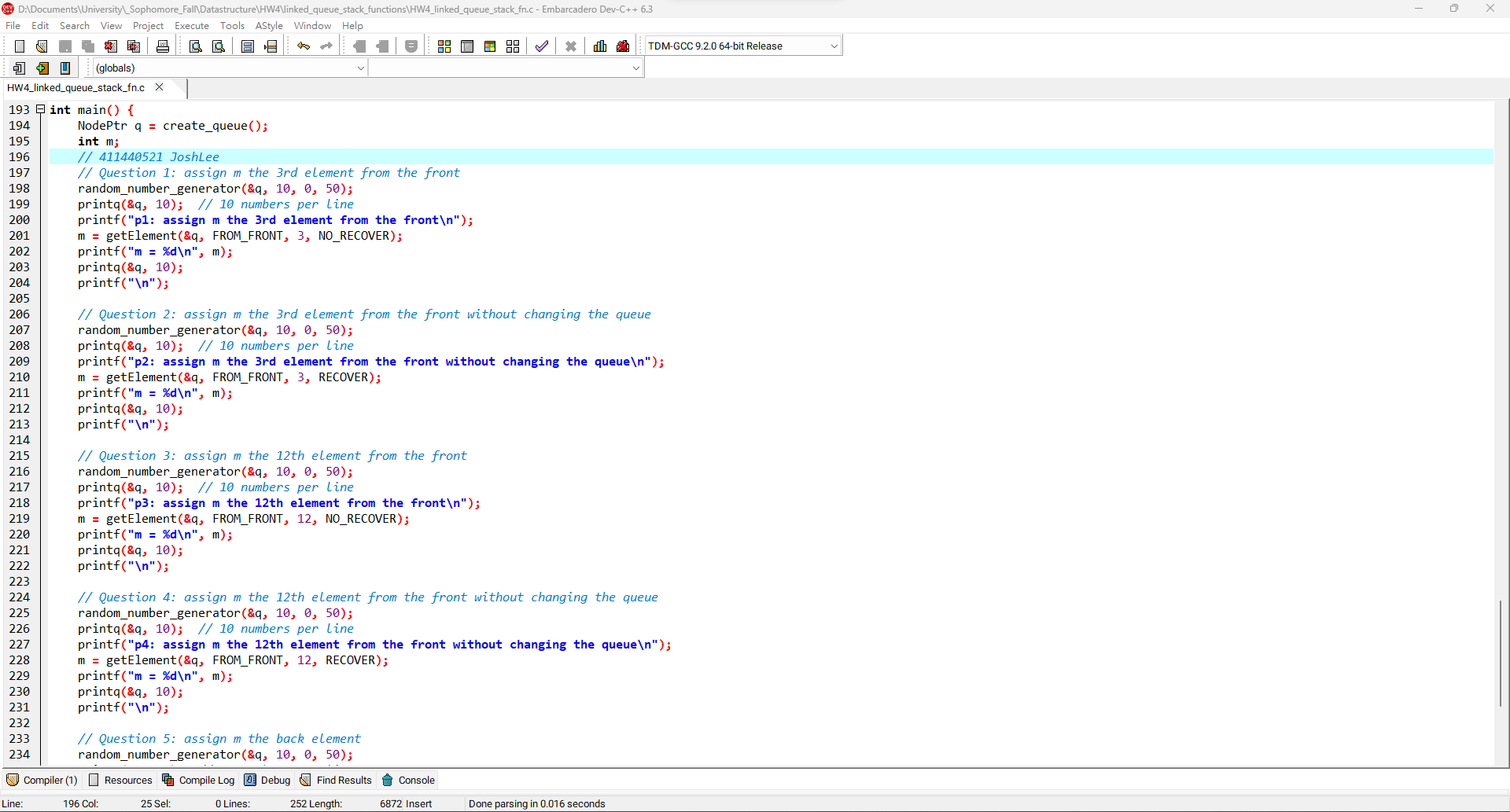
1

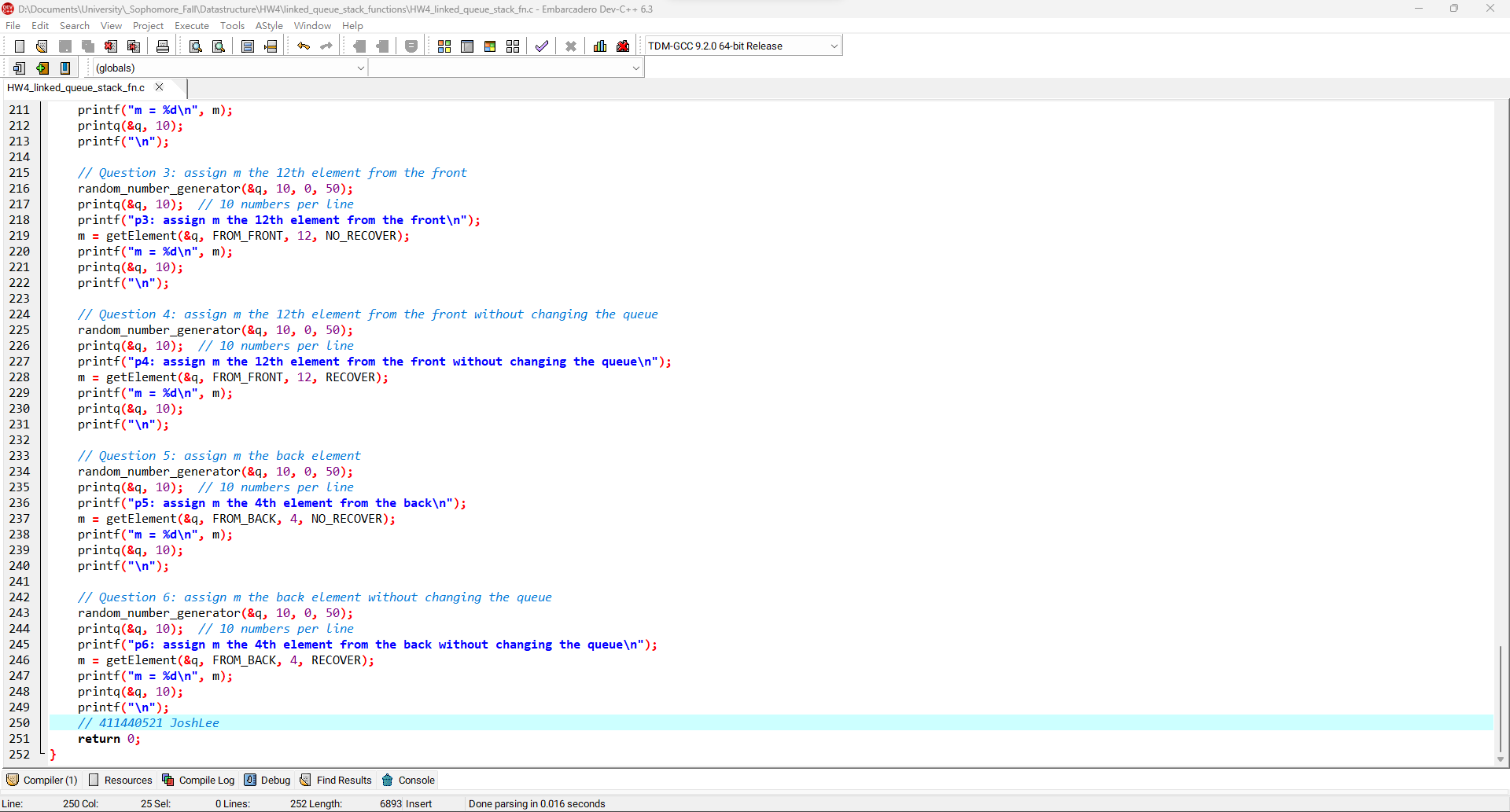
2

3

4

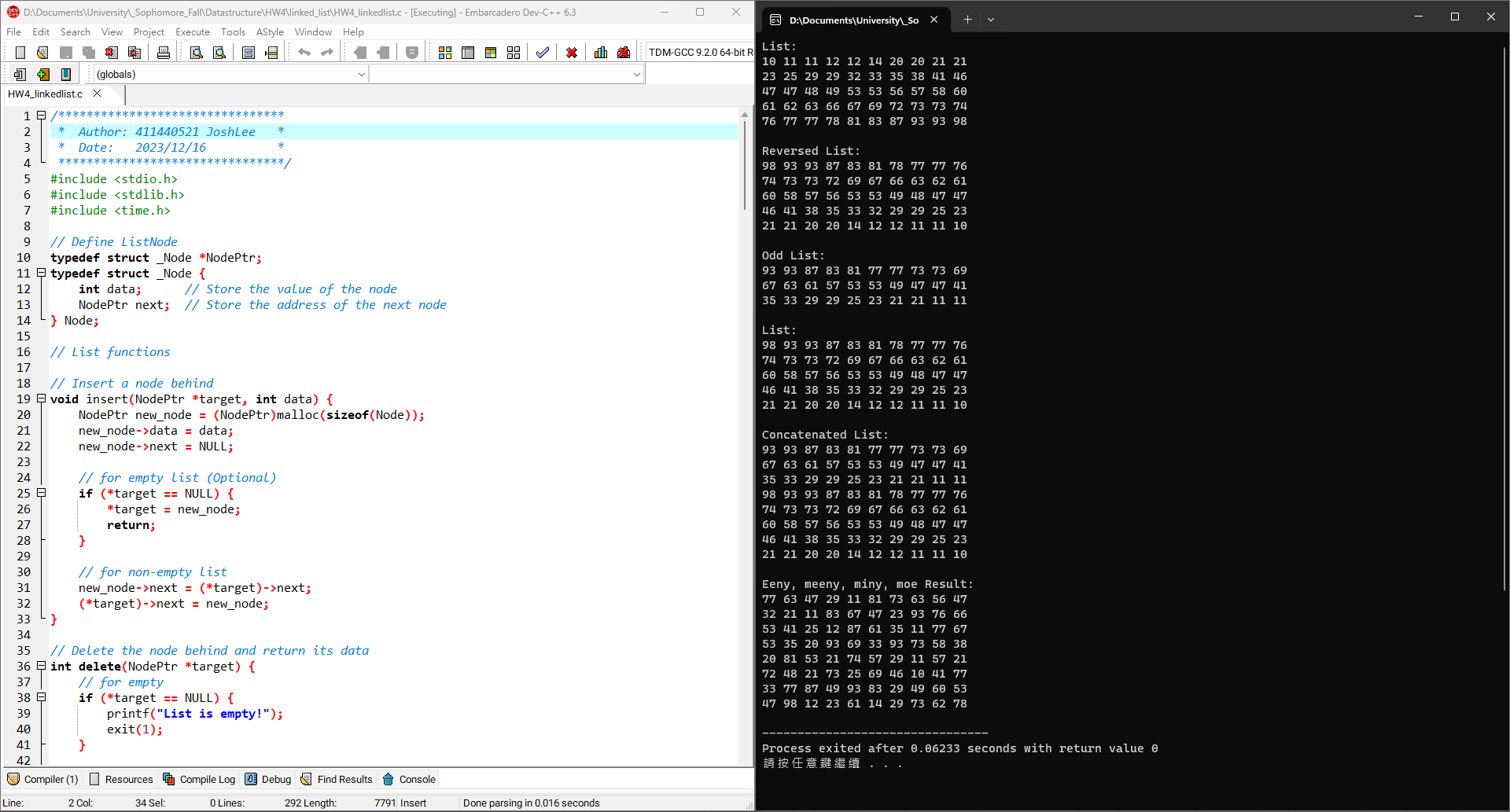
5

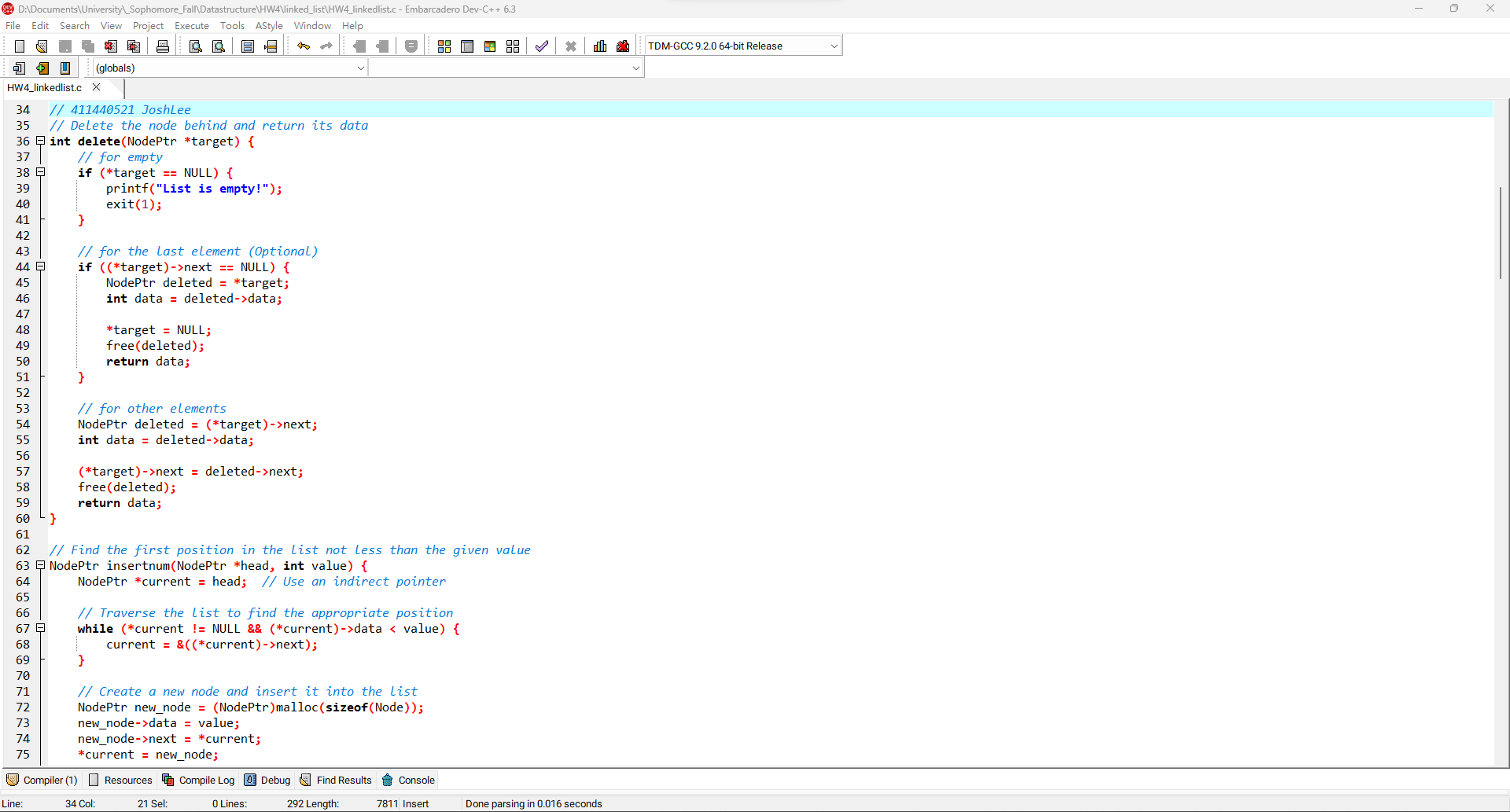
6

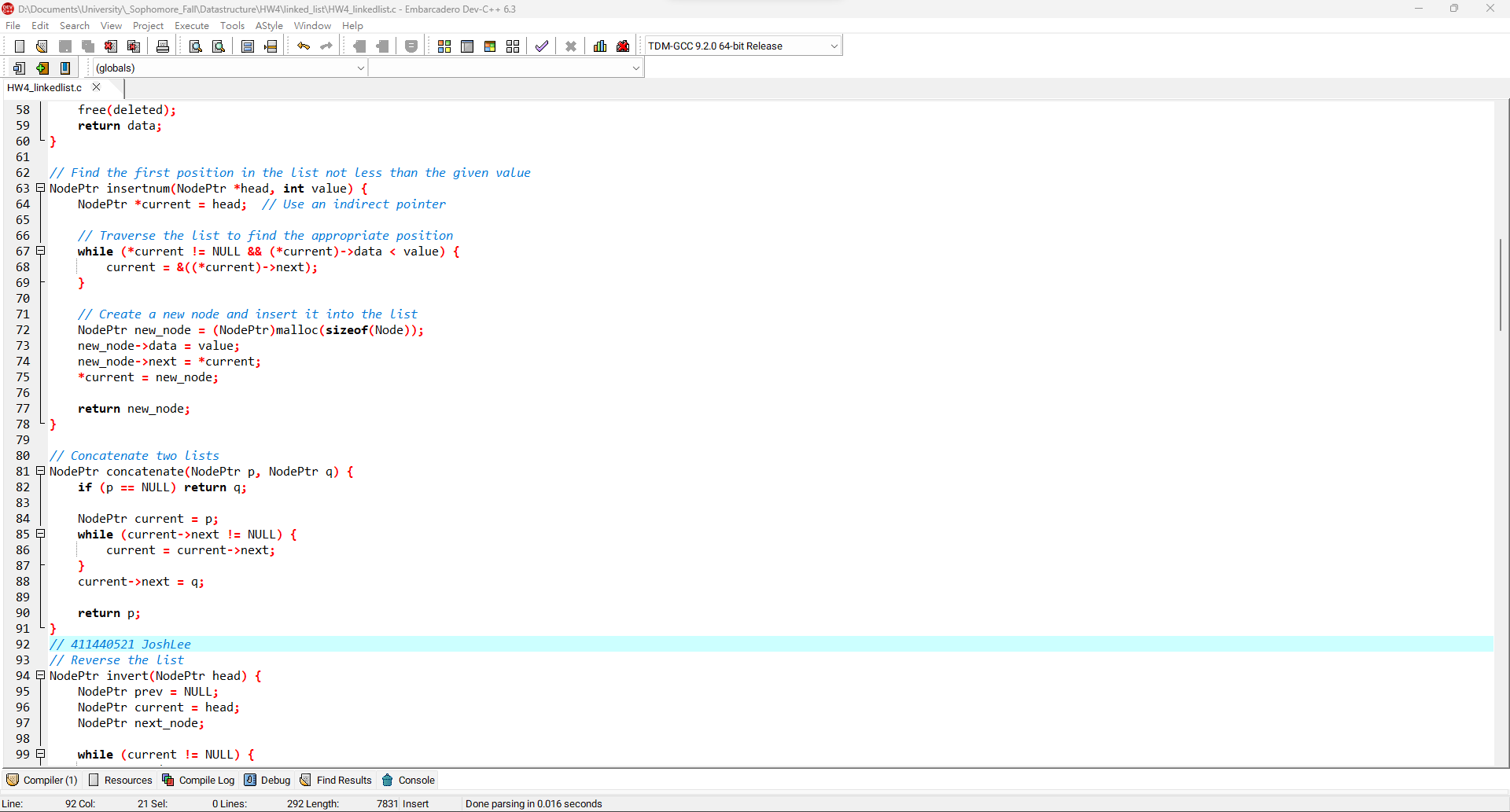
7

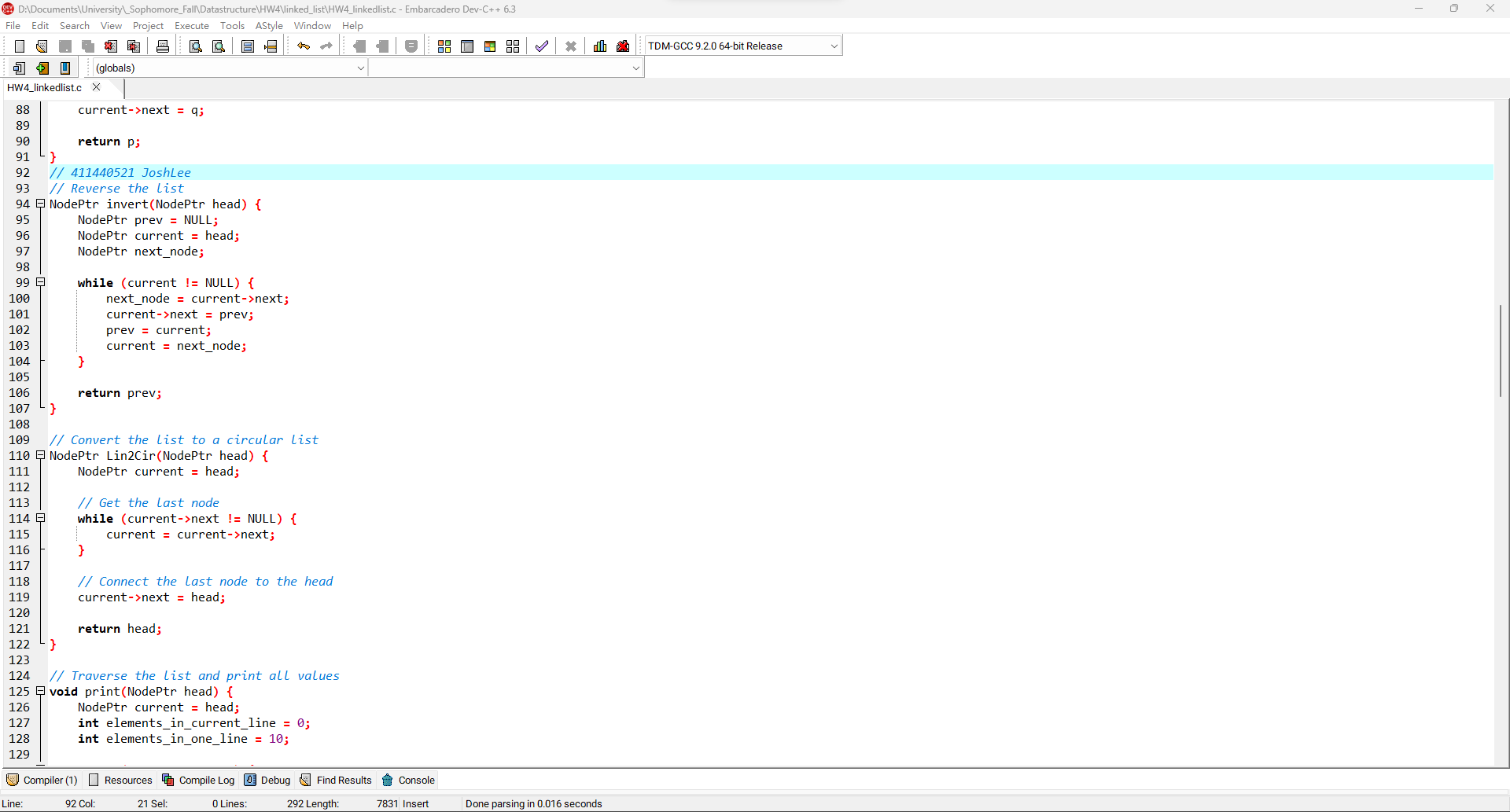
**LinkedQueue (Using LinkedStack Functions) 程式碼:**

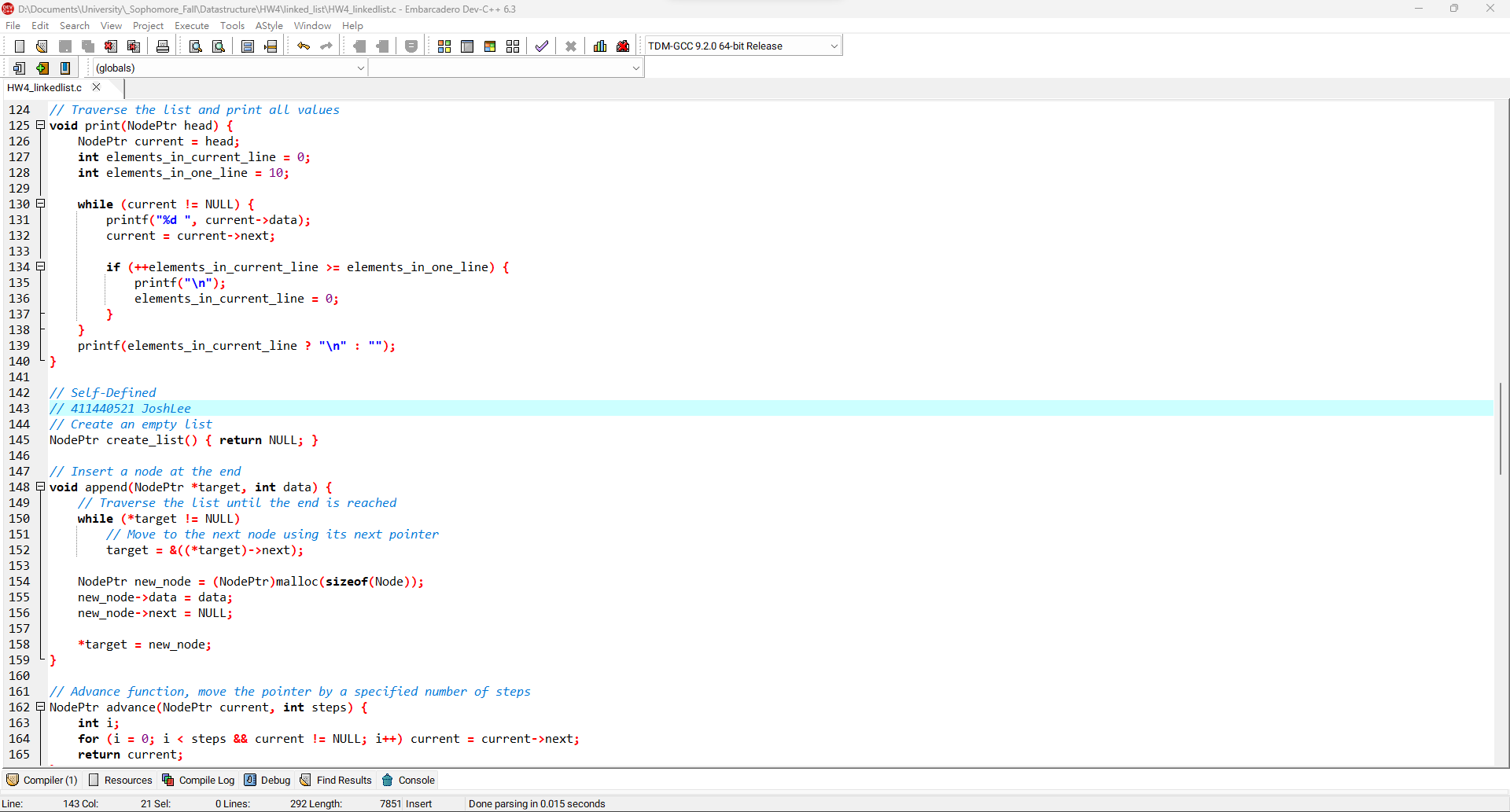
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  *// Enumeration for flags*  typedef enum {  FROM\_FRONT,  FROM\_BACK  } Direction;  typedef enum {  RECOVER,  NO\_RECOVER  } RecoveryOption;  *// Define Node*  typedef struct \_Node \*NodePtr;  typedef struct \_Node {  int data;  NodePtr link;  } Node;  *// LinkedStack functions*  NodePtr create\_stack() {  return NULL;  }  int emptys(NodePtr \*top) {  return \*top == NULL;  }  void pushs(NodePtr \*top, int data) {  NodePtr new\_node = (NodePtr)malloc(sizeof(Node));  if (new\_node == NULL) {  printf("Memory allocation error!\n");  exit(EXIT\_FAILURE);  }  new\_node->data = data;  new\_node->link = \*top;  \*top = new\_node;  }  NodePtr pops(NodePtr \*top) {  if (emptys(top)) {  printf("Stack is empty!\n");  exit(1);  }  NodePtr popped = \*top;  \*top = popped->link;  return popped;  }  void prints(NodePtr \*top, int elements\_in\_one\_line) {  NodePtr current = \*top;  int elements\_in\_current\_line = 0;  while (current != NULL) {  printf("%d ", current->data);  current = current->link;  if (++elements\_in\_current\_line >= elements\_in\_one\_line) {  printf("\n");  elements\_in\_current\_line = 0;  }  }  printf(elements\_in\_current\_line ? "\n" : "");  }  void free\_stack(NodePtr \*s) {  while (!emptys(s)) {  NodePtr popped = pops(s);  free(popped);  }  }  *// LinkedQueue functions (Using LinkedStack Functions)*  NodePtr create\_queue() {  return create\_stack();  }  int emptyq(NodePtr \*front) {  return emptys(front);  }  void addq(NodePtr \*front, int data) {  NodePtr temp\_s = create\_stack();  *// Reverse the order of elements in the stack*  while (!emptys(front)) {  NodePtr popped = pops(front);  pushs(&temp\_s, popped->data);  free(popped);  }  *// Add the new element at the top*  pushs(front, data);  *// Restore the original order of elements*  while (!emptys(&temp\_s)) {  NodePtr popped = pops(&temp\_s);  pushs(front, popped->data);  free(popped);  }  }  NodePtr deleteq(NodePtr \*front) {  if (emptys(front)) {  printf("Queue is empty!\n");  exit(1);  }  return pops(front);  }  void printq(NodePtr \*front, int elements\_in\_one\_line) {  prints(front, elements\_in\_one\_line);  }  void free\_queue(NodePtr \*front) {  free\_stack(front);  }  *// Question*  void random\_number\_generator(NodePtr \*front, int range, int offset, int len) {  srand(time(NULL));  while (!emptyq(front)) deleteq(front);  for (int i = 0; i < len; i++) addq(front, rand() % range + offset);  }  *// Helper function to reverse the queue*  void reverse\_queue(NodePtr \*q) {  if (emptyq(q)) {  return;  }  int fr = deleteq(q)->data;  reverse\_queue(q);  addq(q, fr);  }  *// Helper function to recover elements to the original stack*  void recover\_elements(NodePtr \*q, NodePtr \*temp\_q) {  while (!emptyq(temp\_q)) {  NodePtr deleted = deleteq(temp\_q);  addq(q, deleted->data);  free(deleted);  }  }  *// Helper function to get an element from the stack*  int getElement(NodePtr \*front, Direction direction, int position, RecoveryOption recovery) {  NodePtr temp\_q = create\_queue();  int i, deleted\_value;  if (direction == FROM\_FRONT) {  for (i = 0; i < position; i++) {  NodePtr deleted = deleteq(front);  addq(&temp\_q, deleted->data);  deleted\_value = deleted->data;  free(deleted);  }  reverse\_queue(front);  reverse\_queue(&temp\_q);  if (recovery == RECOVER) recover\_elements(front, &temp\_q);  reverse\_queue(front);  } else {  reverse\_queue(front); *// reverse q to delete the back n-th element*  for (i = 0; i < position; i++) {  NodePtr deleted = deleteq(front);  addq(&temp\_q, deleted->data);  deleted\_value = deleted->data;  free(deleted);  }  reverse\_queue(front); *// reverse q to recover\_elements to the original direction*  reverse\_queue(&temp\_q);  if (recovery == RECOVER) recover\_elements(front, &temp\_q);  }  free\_queue(&temp\_q);  return deleted\_value;  }  int main() {  NodePtr q = create\_queue();  int m;  *// Question 1: assign m the 3rd element from the front*  random\_number\_generator(&q, 10, 0, 50);  printq(&q, 10); *// 10 numbers per line*  printf("p1: assign m the 3rd element from the front\n");  m = getElement(&q, FROM\_FRONT, 3, NO\_RECOVER);  printf("m = %d\n", m);  printq(&q, 10);  printf("\n");  *// Question 2: assign m the 3rd element from the front without changing the queue*  random\_number\_generator(&q, 10, 0, 50);  printq(&q, 10); *// 10 numbers per line*  printf("p2: assign m the 3rd element from the front without changing the queue\n");  m = getElement(&q, FROM\_FRONT, 3, RECOVER);  printf("m = %d\n", m);  printq(&q, 10);  printf("\n");  *// Question 3: assign m the 12th element from the front*  random\_number\_generator(&q, 10, 0, 50);  printq(&q, 10); *// 10 numbers per line*  printf("p3: assign m the 12th element from the front\n");  m = getElement(&q, FROM\_FRONT, 12, NO\_RECOVER);  printf("m = %d\n", m);  printq(&q, 10);  printf("\n");  *// Question 4: assign m the 12th element from the front without changing the queue*  random\_number\_generator(&q, 10, 0, 50);  printq(&q, 10); *// 10 numbers per line*  printf("p4: assign m the 12th element from the front without changing the queue\n");  m = getElement(&q, FROM\_FRONT, 12, RECOVER);  printf("m = %d\n", m);  printq(&q, 10);  printf("\n");  *// Question 5: assign m the back element*  random\_number\_generator(&q, 10, 0, 50);  printq(&q, 10); *// 10 numbers per line*  printf("p5: assign m the 4th element from the back\n");  m = getElement(&q, FROM\_BACK, 4, NO\_RECOVER);  printf("m = %d\n", m);  printq(&q, 10);  printf("\n");  *// Question 6: assign m the back element without changing the queue*  random\_number\_generator(&q, 10, 0, 50);  printq(&q, 10); *// 10 numbers per line*  printf("p6: assign m the 4th element from the back without changing the queue\n");  m = getElement(&q, FROM\_BACK, 4, RECOVER);  printf("m = %d\n", m);  printq(&q, 10);  printf("\n");  return 0;  } |

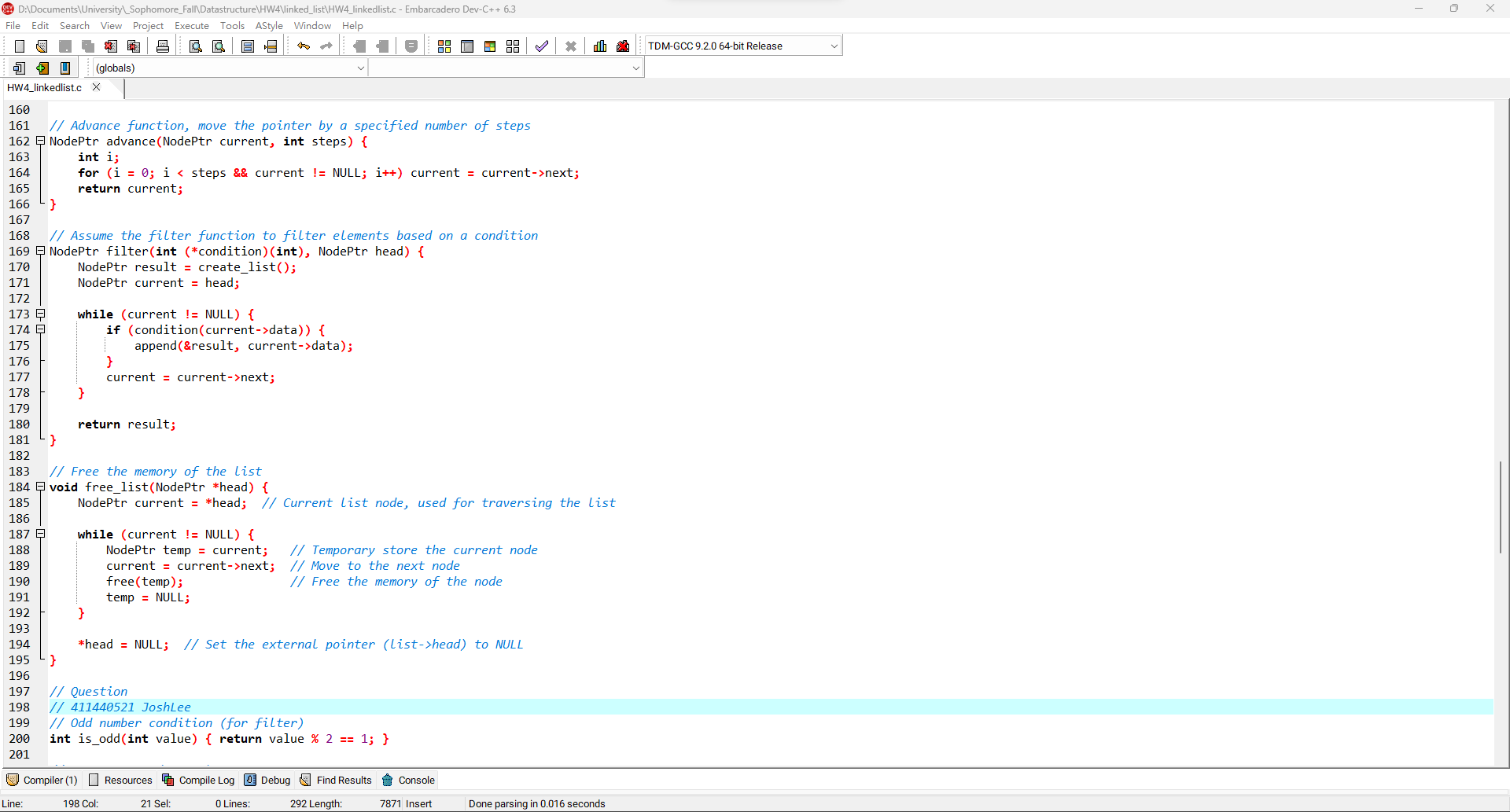
1

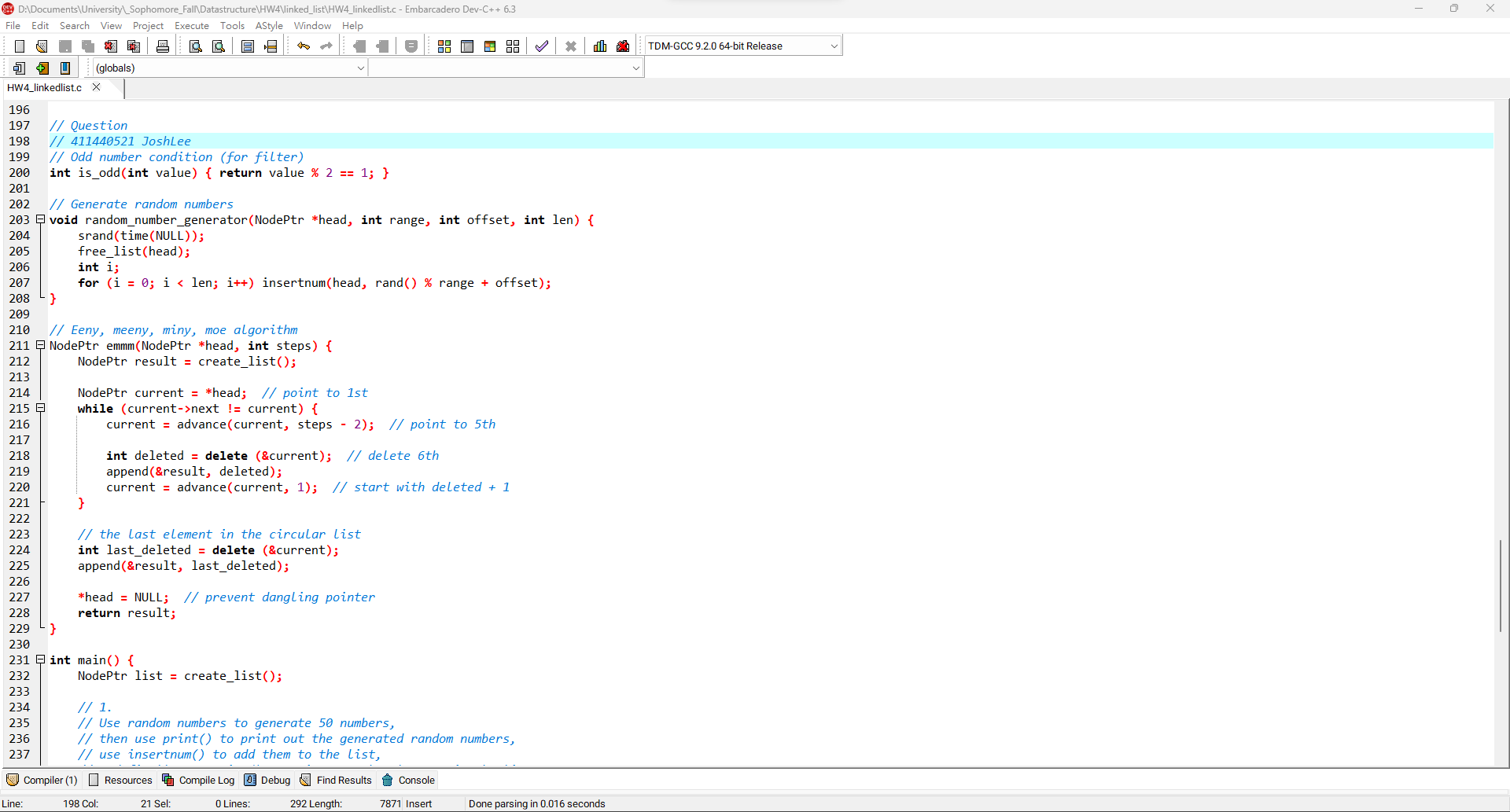
**2**

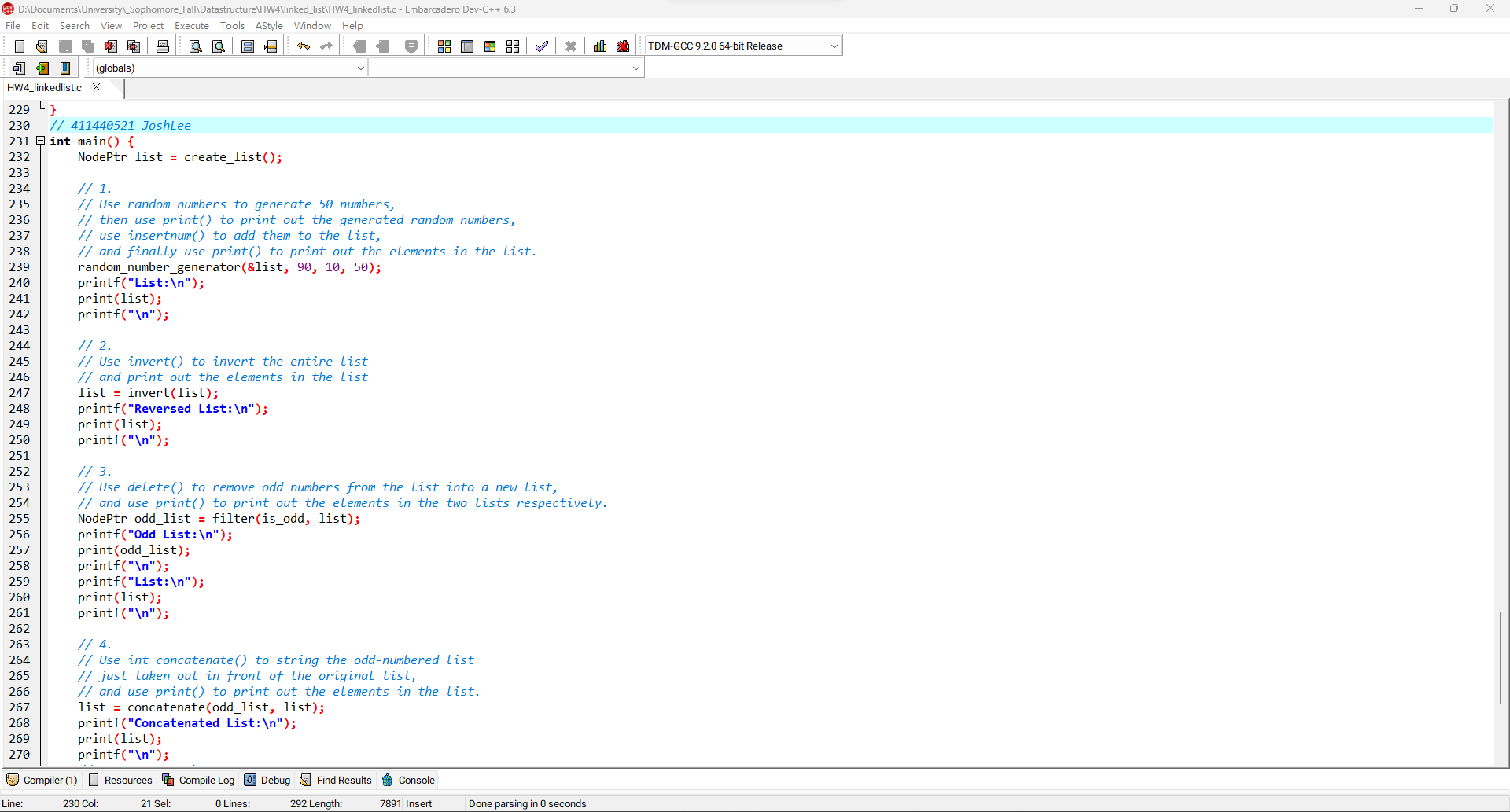
3

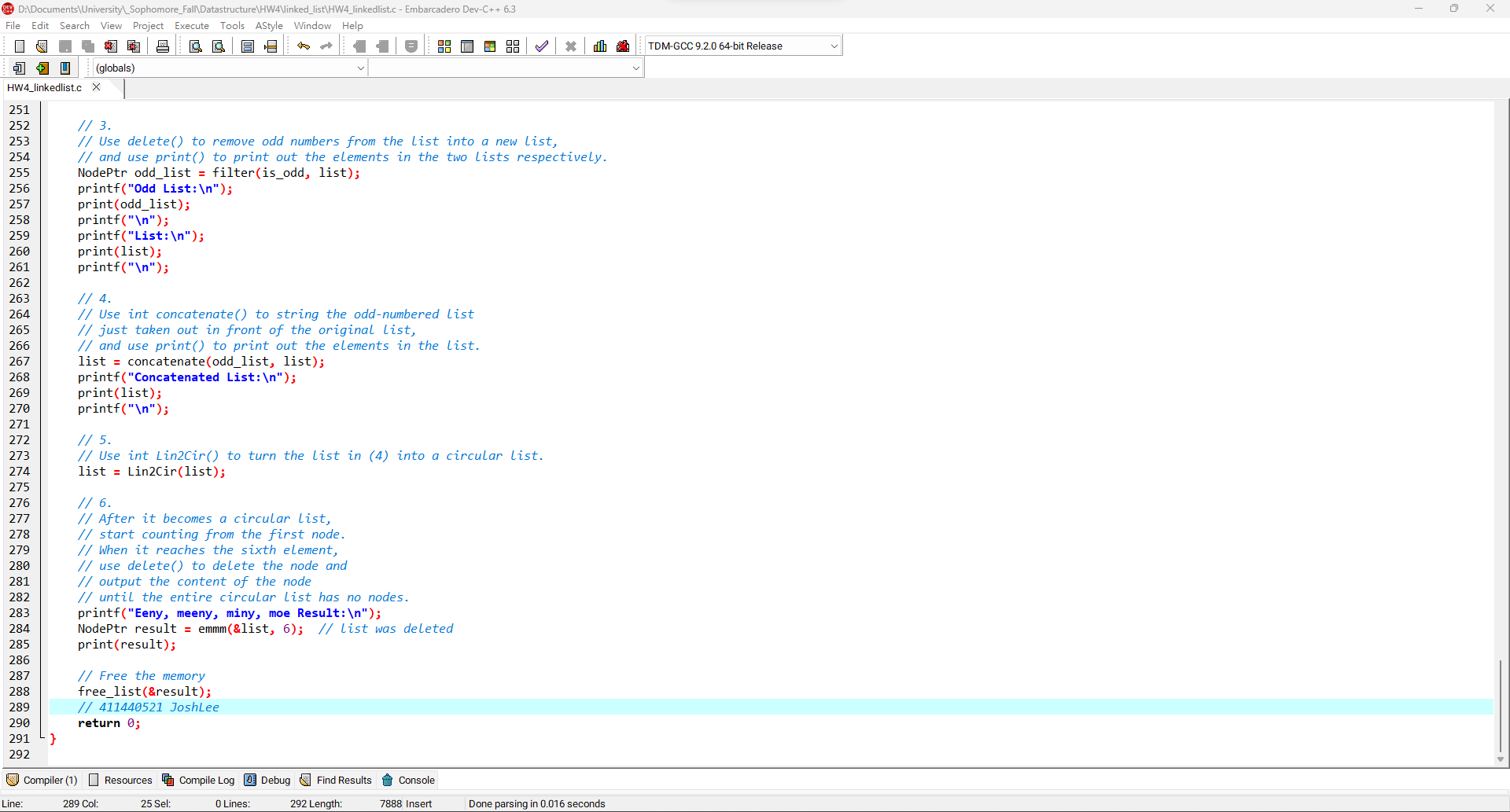
**4**

**5**

**6**

**7**

**8**

**9**

**LinkedList程式碼:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  *// Define ListNode*  typedef struct \_Node \*NodePtr;  typedef struct \_Node {  int data; *// Store the value of the node*  NodePtr next; *// Store the address of the next node*  } Node;  *// List functions*  *// Insert a node behind*  void insert(NodePtr \*target, int data) {  NodePtr new\_node = (NodePtr)malloc(sizeof(Node));  new\_node->data = data;  new\_node->next = NULL;  *// for empty list (Optional)*  if (\*target == NULL) {  \*target = new\_node;  return;  }  *// for non-empty list*  new\_node->next = (\*target)->next;  (\*target)->next = new\_node;  }  *// 411440521 JoshLee*  *// Delete the node behind and return its data*  int delete(NodePtr \*target) {  *// for empty*  if (\*target == NULL) {  printf("List is empty!");  exit(1);  }  *// for the last element (Optional)*  if ((\*target)->next == NULL) {  NodePtr deleted = \*target;  int data = deleted->data;  \*target = NULL;  free(deleted);  return data;  }  *// for other elements*  NodePtr deleted = (\*target)->next;  int data = deleted->data;  (\*target)->next = deleted->next;  free(deleted);  return data;  }  *// Find the first position in the list not less than the given value*  NodePtr insertnum(NodePtr \*head, int value) {  NodePtr \*current = head; *// Use an indirect pointer*  *// Traverse the list to find the appropriate position*  while (\*current != NULL && (\*current)->data < value) {  current = &((\*current)->next);  }  *// Create a new node and insert it into the list*  NodePtr new\_node = (NodePtr)malloc(sizeof(Node));  new\_node->data = value;  new\_node->next = \*current;  \*current = new\_node;  return new\_node;  }  *// Concatenate two lists*  NodePtr concatenate(NodePtr p, NodePtr q) {  if (p == NULL) return q;  NodePtr current = p;  while (current->next != NULL) {  current = current->next;  }  current->next = q;  return p;  }  *// 411440521 JoshLee*  *// Reverse the list*  NodePtr invert(NodePtr head) {  NodePtr prev = NULL;  NodePtr current = head;  NodePtr next\_node;  while (current != NULL) {  next\_node = current->next;  current->next = prev;  prev = current;  current = next\_node;  }  return prev;  }  *// Convert the list to a circular list*  NodePtr Lin2Cir(NodePtr head) {  NodePtr current = head;  *// Get the last node*  while (current->next != NULL) {  current = current->next;  }  *// Connect the last node to the head*  current->next = head;  return head;  }  *// Traverse the list and print all values*  void print(NodePtr head) {  NodePtr current = head;  int elements\_in\_current\_line = 0;  int elements\_in\_one\_line = 10;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  if (++elements\_in\_current\_line >= elements\_in\_one\_line) {  printf("\n");  elements\_in\_current\_line = 0;  }  }  printf(elements\_in\_current\_line ? "\n" : "");  }  *// Self-Defined*  *// 411440521 JoshLee*  *// Create an empty list*  NodePtr create\_list() { return NULL; }  *// Insert a node at the end*  void append(NodePtr \*target, int data) {  *// Traverse the list until the end is reached*  while (\*target != NULL)  *// Move to the next node using its next pointer*  target = &((\*target)->next);  NodePtr new\_node = (NodePtr)malloc(sizeof(Node));  new\_node->data = data;  new\_node->next = NULL;  \*target = new\_node;  }  *// Advance function, move the pointer by a specified number of steps*  NodePtr advance(NodePtr current, int steps) {  int i;  for (i = 0; i < steps && current != NULL; i++) current = current->next;  return current;  }  *// Assume the filter function to filter elements based on a condition*  NodePtr filter(int (\*condition)(int), NodePtr head) {  NodePtr result = create\_list();  NodePtr current = head;  while (current != NULL) {  if (condition(current->data)) {  append(&result, current->data);  }  current = current->next;  }  return result;  }  *// Free the memory of the list*  void free\_list(NodePtr \*head) {  NodePtr current = \*head; *// Current list node, used for traversing the list*  while (current != NULL) {  NodePtr temp = current; *// Temporary store the current node*  current = current->next; *// Move to the next node*  free(temp); *// Free the memory of the node*  temp = NULL;  }  \*head = NULL; *// Set the external pointer (list->head) to NULL*  }  *// Question*  *// 411440521 JoshLee*  *// Odd number condition (for filter)*  int is\_odd(int value) { return value % 2 == 1; }  *// Generate random numbers*  void random\_number\_generator(NodePtr \*head, int range, int offset, int len) {  srand(time(NULL));  free\_list(head);  int i;  for (i = 0; i < len; i++) insertnum(head, rand() % range + offset);  }  *// Eeny, meeny, miny, moe algorithm*  NodePtr emmm(NodePtr \*head, int steps) {  NodePtr result = create\_list();  NodePtr current = \*head; *// point to 1st*  while (current->next != current) {  current = advance(current, steps - 2); *// point to 5th*  int deleted = delete (&current); *// delete 6th*  append(&result, deleted);  current = advance(current, 1); *// start with deleted + 1*  }  *// the last element in the circular list*  int last\_deleted = delete (&current);  append(&result, last\_deleted);  \*head = NULL; *// prevent dangling pointer*  return result;  }  *// 411440521 JoshLee*  int main() {  NodePtr list = create\_list();  *// 1.*  *// Use random numbers to generate 50 numbers,*  *// then use print() to print out the generated random numbers,*  *// use insertnum() to add them to the list,*  *// and finally use print() to print out the elements in the list.*  random\_number\_generator(&list, 90, 10, 50);  printf("List:\n");  print(list);  printf("\n");  *// 2.*  *// Use invert() to invert the entire list*  *// and print out the elements in the list*  list = invert(list);  printf("Reversed List:\n");  print(list);  printf("\n");  *// 3.*  *// Use delete() to remove odd numbers from the list into a new list,*  *// and use print() to print out the elements in the two lists respectively.*  NodePtr odd\_list = filter(is\_odd, list);  printf("Odd List:\n");  print(odd\_list);  printf("\n");  printf("List:\n");  print(list);  printf("\n");  *// 4.*  *// Use int concatenate() to string the odd-numbered list*  *// just taken out in front of the original list,*  *// and use print() to print out the elements in the list.*  list = concatenate(odd\_list, list);  printf("Concatenated List:\n");  print(list);  printf("\n");  *// 5.*  *// Use int Lin2Cir() to turn the list in (4) into a circular list.*  list = Lin2Cir(list);  *// 6.*  *// After it becomes a circular list,*  *// start counting from the first node.*  *// When it reaches the sixth element,*  *// use delete() to delete the node and*  *// output the content of the node*  *// until the entire circular list has no nodes.*  printf("Eeny, meeny, miny, moe Result:\n");  NodePtr result = emmm(&list, 6); *// list was deleted*  print(result);  *// Free the memory*  free\_list(&result);  *// 411440521 JoshLee*  return 0;  } |