ASSESSMENT COVER SHEET

Student Name	Shane Reynolds
Student ID	262538
Assessment Title	Quiz 2
Unit Number and Title	HIT365
Lecturer/Tutor	Kai Wang
Date Submitted	18/4/2014
Date Received	18/4/2014

KEEP A COPY

Please be sure to make a copy of your work. If you have submitted assessment work electronically make sure you have a backup copy.

PLAGIARISM

Plagiarism is the presentation of the work of another without acknowledgement. Students may use a limited amount of information and ideas expressed by others but this use must be identified by appropriate referencing.

CONSEQUENCES OF PLAGIARISM

Plagiarism is misconduct as defined under the Student Conduct By-Laws. The penalties associated with plagiarism are designed to impose sanctions on offenders that reflect the seriousness of the University's commitment to academic integrity.

I declare that all material in this assessment is my own work except where there is a clear acknowledgement and reference to the work of others. I have read the University's Academic and Scientific Misconduct Policy and understand its implications.*

http://www.cdu.edu.au/governance/documents/AcademicandScientificMisconductPolicyv1.03Jan2011.pdf.

Signed.....Date.....13/4/2014.....

*** By submitting this assignment and cover sheet electronically, in whatever form you are deemed to have made the declaration set out above.**

ASSIGNMENT 2

HIT365 – C programming

This assignment is worth 5% of the assessment for this unit. Late submissions will not be accepted unless prior arrangements have been made.

Show all working and explanations. Remember to write sentences.

Successfully completion of tutorial exercises is worth 20% of your mark

QUESTION 1 (30 MARKS)

Fix or describe the error in each of the following pieces of code:

a)

float *realPtr;

long *intPtr;

intPrt = realPtr;

Response

The principal issue with this piece of code is that the pointers are of different type. You could make both pointers the same type. Also the pointer is not initialized. The following code below rectifies the issues.

```c
/*
 * Question1a.c
 * Created on: 18/04/2014
 * Author: Shane
 */

#include <stdio.h>

int main(void){

	float a = 2.0;
	float *realPtr = &a;
	printf("%p\n", &realPtr);
	printf("%f\n\n", *realPtr);

	float *intPtr = realPtr;
	printf("%p\n", &intPtr);
	printf("%f\n\n", *intPtr);

	return 0;

}
```

I had a couple of problems with this question – why don't the two pointer addresses match? The dereferenced values are the same, but not the addresses that the pointers point to.

Also are you able to type cast a pointer when assigning a value to it? I tried this and I couldn't get it to work.

b)

float x = 29.23;

float xPrt = &x;

printf(“%f\n”,xPtr)

Response

The principal issue with this piece of code is on the second line. The first line sees the value of 29.23 stored to a variable of type float. The second line attempts to store the address of the variable x to a pointer xPtr, however, there is a syntactic error here. When specifying a pointer we need to use the * operator. Finally, there are a couple of small syntactic errors on the third line. The variable xPrt is not written correctly – a consistent variable name needs to be chosen. Also, the printf statement doesn’t have a semi-colon. Finally, to print the value that the pointer points to we need to dereference the pointer. Corrected code is given below.

```
#include <stdio.h>

int main(void){

	float x = 29.23;
	float *xPtr = &x;
	printf("%f",*xPtr);

	return 0;
}
```

c)

int *x,y;

x=y;

Response

The code written above specifies two integer type variables: one as a pointer and one as a normal variable. The second line then assigns the value of y to the pointer. This is wrong. The code may have been intended to assign the address of the variable y to the pointer x in which case you would need to use the & operator. Code with no error is provided below.

```
#include <stdio.h>

int main(void){

	int *x,y;

	x=&y;

	printf("%p", &x);

	return 0;
}
```

QUESTION 2 (30 MARKS)

In minor orders a number of simple operations are allowed. These are

- Vertex deletion
- Edge Contraction

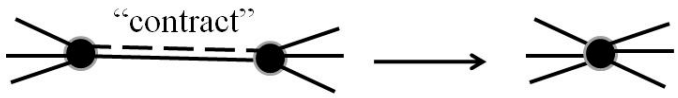

Figure 1 Edge Contraction

An easy way of understanding an edge *e*, being contracted is to just imagine *e* being lifted (or shrunk) until its adjacent vertices join together and them deleting the edge. (See Figure 1); however, for those interested in a definition, edge Contraction is defined as:
Let G be an undirected graph.
Let $e \in E(G)$ be an edge of G.
Then the graph obtained by contracting e in G, denoted by G/e, is the graph H defined by:

$V(H)=(V(G)\setminus e)\cup\{v\}$
$E(H)=\{f \in E(G): f \cap e = \emptyset\}\cup\{ uv: \exists f \in E(G): u \in f \setminus e, f \cap e \neq \emptyset\}$
where $v \notin V(G)$ is a new vertex.

Write a function that takes as input a graph and an edge e and contract e. The edge should be passed by value and the graph should be passed by reference.

To do this:

- Write a test function.
- Write you contraction function stub only
- Run you test and see that it fails
- Now write you contraction function.
- Run your test
- Look at you function and see if there is any way to improve you coding
- What is the complexity of your contraction function

Question 2 Response

```c
/*
 * DynamicQuestion2.c
 * Created on: 17/04/2014
 * Author: Shane
 */

/* Preprocessing directives */
#include <stdio.h>
#include <stdlib.h>


/* Function prototypes */
void printMat(int **array, int nrow, int ncol);
int **removeEdge(int **array, int nrow, int ncol, int v1, int v2);
void test(int **array, int nrow, int ncol);


int main(void){

	/*************************************************************/

	/* This and the real results in the test code are the only
	 * parts of the code that need changing - everything else is
	 * dynamically allocated */

	int data[6][6] =    {	{0,1,1,0,1,1},
				{1,0,1,1,1,0},
				{1,1,0,1,1,0},
				{0,1,1,0,1,1},
				{1,1,1,1,0,1},
				{1,0,0,1,1,0}};

	/*************************************************************/

	/* Variables for main function specified  */
	int **graphOld;
	int i,j;
	int vertex1, vertex2;
	int nrow = sizeof(data) / sizeof(data[0]);
	int ncol = nrow;

	/* Dynamically allocated memory to store the existing adjacency matrix.
	 * This data structure is a array of pointers which make up the rows.
	 * Each pointer in the array of pointers each point to an array of
	 * pointers which makes up the dynamic 2D array  */
	graphOld = calloc(nrow, sizeof(int*));

	for (i = 0; i < nrow; i++){
		graphOld[i] = calloc(ncol, sizeof(int));
	}

	/* Populate our dynamically allocated 2D array with the actual
	 * adjacency matrix */
	for (i = 0; i < nrow; i++){
		for (j = 0; j < ncol; j++){
			graphOld[i][j] = data[i][j];
		}
	}

	/* Print the original adjacency matrix */
	printf("\nOriginal adjacency matrix:\n\n");
```

```
	printMat(graphOld, nrow, ncol);

	/* The edge to be contracted can be thought of as two vertices -
	 * that is the edge can be defined by two vertices. This code
	 * prompts the user for the two vertices which make up the
	 * edge they want to contract */
	printf("Enter the number of the first vertex of the edge: ");
	scanf("%d", &vertex1);
	printf("Enter the number of the second vertex of the edge: ");
	scanf("%d", &vertex2);

	printf("\n");

	/* Call to the function which contracts the graph. The result is
	 * returned to an pointer which is initialised in the line */
	int **graphPtr = removeEdge(graphOld, nrow, ncol, vertex1, vertex2);

	/* Print the contracted graph */
	printf("Contracted adjacency matrix:\n\n");
	printMat(graphPtr, nrow-1, ncol-1);

	/* A test function to ensure that the correct results are
	 * achieved */
	test(graphPtr, nrow, ncol);

	return 0;
}

/* This function receives a dynamically allocated 2D array and
 * prints it */
void printMat(int **array, int nrow, int ncol){
	int i, j;

	for (i = 0; i < nrow; i++){
		for (j = 0; j < ncol; j++){
			printf("%d ", array[i][j]);
		}
		printf("\n\n");
	}

}

/* This function receives a 2D array as input, along with the two
 * vertices that need to be contracted to one. It outputs a pointer
 * to an array. */
int **removeEdge(int **array, int nrow, int ncol, int v1, int v2){

	/* specify and initialise variables */
	int **graphNew;
	int i, j;
	int m = 0;
	int n = 0;

	/* Create a place in the heap memory (initialised with zeros)
	 * for the contracted graph to be stored */
	graphNew = calloc(nrow-1, sizeof(int*));

	for (i = 0; i < nrow-1; i++){
		graphNew[i] = calloc(ncol-1, sizeof(int));
	}

	/* This piece of code takes the columns which aren't the contracting vertexes
	 * and shifts them to the left and takes the rows which aren't the
```

```
	 * contracting vertexes and shifts them up NOTE: there will be
	 * a final row and final column which represents the contracted vertex. */
	for (i = 0; i < nrow; i++){
		n = 0;
		if (i != (v1-1) && i != (v2-1)){
			for (j = 0; j < ncol; j++){
				if (j != (v1-1) && j != (v2-1)){
					graphNew[m][n] = array[i][j];
					n += 1;
				}
			}
			m += 1;
		}
	}

	/* This code completes the contracted vertex on the final row and column
	 * NOTE: this only works for undirected graphs - it takes advantage of the
	 * symmetry in the adjacency matrix to complete the contracted adjacency
	 * matrix. */
	m = 0;

	for (j = 0; j < ncol; j++){
		if (j != (v1-1) && j != (v2-1)){
			if (array[v1-1][j] == 1 || array[v2-1][j] == 1){
				graphNew[nrow-2][m] = 1;
				graphNew[m][ncol-2] = 1;
			}
			m += 1;
		}
	}

	/* Finally the contracted vertex cannot access itself */
	graphNew[nrow-2][ncol-2] = 0;

	return graphNew;

}

/* Basic test which ensures that the results from the code are
 * correct */
void test(int **array, int nrow, int ncol){

	/*************************************************************/
	/* User needs to put the actual results in to a 2D array
	 * for the test function to accurately test the output of
	 * the removeEdge() function */

	int data[5][5] = {  {0,1,1,1,1},
			    {1,0,1,0,1},
			    {1,1,0,1,1},
			    {1,0,1,0,1},
			    {1,1,1,1,0}  };

	/*************************************************************/

	int i,j;
	int pass = 1;

	for (i = 0; i < (nrow-2); i++){
		for (j = 0; j < (ncol-2); j++){
			if (array[i][j] != data[i][j]){
				pass = 0;
				break;
```

```
                        }
                }
        }

        if (pass == 1){
                printf("Test passed.\n");
        } else {
                printf("Test failed.\n");
        }
}
```