Tutorial - Week 8 Solution

Objectives: To practice with
- Pointers
- Dynamic Memory Allocation

1. **Assuming the following declaration of the variables `cType` and `voltage`. Declare the corresponding number of pointer type variables and assign them with the memory addresses of `cType` and `voltage`.**

```
char cType;
float voltage;
```

```
char *cptr;
float *fptr;

cptr = &cType;
fpter = &voltage;
```

2. **Can one pointer be used to access several variables (one at a time) of the same type? Provide an example.**

```
float f1, f2, f3;
float *ptr;
ptr = &f1;
ptr = &f2;
ptr = &f3;
```

3. **Provide an example of changing a variable value using a pointer and the indirection operator.**

```
int width;
int *pt;
pt = &width;
*pt = 360;/* equivalent to width = 360  */
```

4. **What are the values of `ptrI` and `ptrC`?**

```
int *ptrI = 1000;
char *ptrC = 1000;

ptrI += 2;
ptrC += 2;
```

Explain the results.

```
/* ptrI =  1000 + 2*sizeof(int) = 1008    */
/* ptrC =  1000 + 2*sizeof(char) = 1002  */
```

IMPORTANT: Never assign a constant address value to a pointer variable as it often leads to memory segmentation. Regardless, the above pointer definitions generate warnings unlike type cast is used , see the lecture notes.

5. **Given the declarations:**

```
int m = 25, n = 77;
char c = '*';
int *itemp;
```

describe the errors in each of the following statements.

```
m = &n;
itemp = m;
*itemp = &c;
```

Answer: The above statements compile with warnings which can all be removed using type cast (given 64 bit system, you'll have to cast to long int instead of int when applicable).However although it compiles, the resulting statements will be illogical, unless the user does it on purpose which goes out of the scope of this subject.

Based on the above, we'll correct the above statement as per pointer variable expected use:

```
m = &n;  /* attempt to assign the address of an integer variable
            to a non-pointer variable. The user might have wanted
            m = n; */

itemp = m;  /* attempt to assign an integer value to a pointer
             variable. The user might have wanted  itemp = &m;


*itemp = &c;   /* attempt to assign an address to an integer
                location. The user might have wanted : *itemp = c;
                (ASCII code will be stored) */
```

Extra: The following statements compile without any warning in 64-bit computer:

```
m =(long int) &n;
itemp =( int *)((long int) m);
*itemp =(long int ) &c;
```

6. **Write a program in C to find the factorial of a given number using pointers.**

Constraint: a function return type can't be a pointer unless the

local pointer **variable is declared as static.** However, we can return a value to the caller via a function pointer parameter.

Thus the prototype of our function is
void findFact(int num1, int *fact);

```
#include <stdio.h>
void findFact(int n,int *f);
int main()
{
          int fact;
          int num1;
          printf("\n\n Pointer : Find the factorial of a
                 given number :\n");
          printf("-------------------------------------\n");

          printf(" Input a number : ");
          scanf("%d",&num1);

          findFact(num1,&fact);
          printf(" The Factorial of %d is : %d
                    \n\n",num1,fact);
          return 0;
         }

void findFact(int n,int *f)
     {
        int i;

       *f =1;
       for(i=1;i<=n;i++)
       *f=*f*i;
     }
```

7. a) What are the values of main function variables x and y at the point marked / * values here */ in the following program?

```
/* nonsense */

void silly(int x);
int main(void)
{
     int x, y;
     x = 10; y = 11;
     silly(x);
     silly(y);            /* values here */
     . . .
}

void silly(int x)
```

```
{
      int y;
      y = x + 2;
      x *= 2;
}
```

```
x is 10, y is 11
```

Call function with PASS by Value. The function has void return type

b)

```
/* nonsense */
void silly(int *x);
int main(void)
{
      int x, y;
      x = 10; y = 11;
      silly(&x);
      silly(&y); /* values here */
      . . .
}
void silly(int *x)
{
      int y;
      y = *x + 2;
      *x = 2 * *x;
}
```

```
x is 20, y is 22
```

With pointers, we implemented FUNCTION CALL with PASS BY REFERENCE, the caller's argument variable can be changed by the called function

Homework: what will be the printed values with the two inserted printf statements to the above program:

```
silly(&x);
printf("%d %d\n", x, y);
silly(&y); /* values here */
printf("%d %d\n", x, y);
```

Run the code to verify the correctness of your answers.

8. What is the output of the following programs?

```c
#include <stdio.h>

int main (void)
{
     int   count = 10, x;
     int   *int_pointer;

     int_pointer = &count;
     x = *int_pointer;

     printf ("count = %i, x = %i\n", count, x);

     return 0;
}
```

```
Count =10, x =10
```

```c
#include <stdio.h>

int main (void)
{
     char  c = 'Q';
     char  *char_pointer = &c;

     printf ("%c %c\n", c, *char_pointer);

     c = '/';
     printf ("%c %c\n", c, *char_pointer);

     *char_pointer = '(';
     printf ("%c %c\n", c, *char_pointer);

     return 0;
}
```

```
QQ
//
((
```

9. Write a statement to execute the following:

a. **Declare a pointer to an integer called `address`.**

b. **Assign the address of a float variable `balance` to the float pointer `temp`.**

c. **Assign the character value '`W`' to the variable pointed to by the char pointer `letter`.**

d. **Declare a pointer to the text string `"Hello"` called `message`.**

e. **Assume float `balance[10][5]`. How can you access `balance[3][1]` using pointers?**

```c
a) int *address;
b) temp = &balance;
c) *letter = 'W';
d) char *message = "Hello";
e) *(balance + (3*5) + 1)
```

10. Allocate memory to store an array of 50 integers and initialize elements of this dynamic array with values equal to their indexes (0 – 49). The code must provide memory allocation error checking.

```
#include<stdlib.h>
int i, *iPtr;
iPtr = malloc(50*sizeof(int));
if( iPtr == NULL )
      return -1;
for(i=0; i<50; i++)
  iPtr[i] = i;       or    *(iPtr +i) = i;
```

11. Assuming that the function `getNextValue()` correctly returns a `float` type value, what is wrong with this code and fix it?

```
int i;
float *flArr;

if( (flArr = calloc(256, sizeof(float))) ==  NULL)
{
   printf(" Memory allocation failure");
   return (-1);
}

for(i=0; i<256; i++)
  *flArr++ = getNextValue();

  free( flArr );
```

Answer:

The above code generates a runtime error (double free or corruption (out)) as the loop statement changes the value of flArr which pointed to the start of the dynamic memory segment allocated by calloc; there free(flArr) generates the runtime exception. Recall and as explained in the lecture notes *flArr++ is equivalent to *(flArr++).

To fix the code, we suggest to replace `*flArr++ = getNextValue();` **with either**

`*(flArr+i) = getNextValue();` **or** `flArr[i] = getNextValue();`

i.e. flArr used in the calloc should not be change for the free(flArr) to be always successful.

12. Define a function

```
char *createEmptyString( int  length);
```

that creates an empty c-string of a specified length. Write a simple `main()` function to test `createEmptyString()`.

Answer:

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

```
#include<string.h>  // this is needed for the memset

char* createEmptyString( int length );

int main (void)
{
    char *newString = NULL;
    newString = createEmptyString( 20 );
    if(newString != NULL)
       printf("%s",newString);

    return 0;
}



char* createEmptyString( int length  )
{
  static char *buffer=NULL;
  buffer=calloc( length + 1,sizeof(char ));
  // we use calloc to set the NULL terminating character of the
    string

    if(buffer != NULL)
       memset(buffer, 'A', length);

    return buffer;
    }
```

IMPORTANT: The keyword static has to be added otherwise, the pointer can be removed from the memory when the function terminates its execution. We added the memset function in order to set the first length bytes of buffer to 'A' , can be any character, so length 'A' will be printed; a NULL character is not printed.

13. Write a sample of code to allocate memory for:

- **one item of type `component` and initialize it**
- **a dynamic array with `SIZE` elements of type `component` and initialize the 3rd element.**

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

#define SIZE 10

typedef struct
{
 char type[7];
 float price;
 int quantity;
} component_t;

component_t *item;
```

```
int main(){

item = malloc(SIZE *sizeof(component_t));
if( item == NULL )
{
    // . .error recovery ..
}

item[2].price = 15.0;
item[2].quantity = 100;
strcpy( item[2].type, "TGZ25" );
printf("%.2f %d %s", (*(item+2)).price,(item+2)-
>quantity,(item+2)->type);

free(item);
}

Alternatively, you can use
component_t *indexer;
indexer=item;
indexer=item+2;
indexer->price = 15.0;
indexer->quantity = 100;
strcpy( indexer->type, "TGZ25" );
printf("%.2f %d %s", (*(item+2)).price,(item+2)-
>quantity,(item+2)->type);
```

14. Differentiate between dynamic and nondynamic data structures.

Dynamic data structures are allocated on the heap as a result of executing calls to memory allocation functions such as malloc and calloc. These structures can grow and shrink as a program executes, and they are referenced through pointers. They remain allocated until explicitly deallocated by a call to free. Non-dynamic data structures are allocated on the stack when the function to which they belong is called and are deallocated when the function returns. Their size is known when the program is compiled--they do not grow and shrink. They are referenced by their names.