Digital Design and Verification

IFYP Program

Pointers & Linked List

Release: 1.0

Date: 15-August-2024

Revision History

Revision Number	Revision Date	Revision By	Nature of Revision	Approved By
1.0	15/08/2024	S. M. Sarmad	First Draft	
1.1	03/03/2025	Hira Sohail	Revision	

TASK # 01:

A pointer is a variable that holds a memory address. Pointers are declared using the asterisk (*) operator:

```
int *p;
```

In the above example, the type of the variable p is int* (usually pronounced "int star" or "int pointer"). Like other variables, it is a fixed reference to an allocated place in memory. Unlike other variables, that memory location itself holds the address of another place in memory. That is what makes it a "pointer;" it "points" to another location.

Until it is initialized, however, it doesn't point to anything in particular. To avoid a lot of headaches later on, pointers should always be intialized when they are declared. If you don't know what it should point to immediately, you can always initialize it to a special value (zero) using the NULL constant:

```
int *p = NULL;
```

Question: How would we initialize p so that it points to x?

WARNING: If you wish to declare multiple pointers on the same line, you must include an asterisk with each of them. For instance:

```
int *p = NULL, *q = NULL;
```

In this course, we strongly recommend declaring multiple pointers on a single line.

To assign pointers so that they point to existing variables, you can use the address-of operator (&) just as we used it earlier to print the address:

```
p = &x;
```

After this line, p is now a pointer to the location that stores the value associated with the variable x. Note that x is NOT a pointer; it is just a variable name.

You can now print the value of p (the pointer) and verify that it is the same as the address of x:

```
printf("Pointer value of p is %p\n", p);
```

Dereferencing pointers

To retrieve the value that a pointer points to, use the "dereference" operator (*). Yes, it's an asterisk, the same as we used to declare the pointer.

In the running example, we know that p is an int* (an "int-pointer" or a "pointer to an int"). We have also initialized it to point to the location of variable x. Thus, we can print the integer that p points to using the following code:

```
printf("Dereferenced value of p is %d, and x = %d\n", *p, x);
```

Question: What would happen if we passed p instead of *p to printf?

Question: What would happen if we dereferenced a pointer that had the value of NULL?

Exercise:

What does the following code print? Step through it as if you were the machine. This is sometimes referred to as "tracing" the code. You may find it helpful to draw diagrams with boxes for memory locations and arrows for pointers.

```
int a =
42; int b
= 7; int c
= 999;
int *t =
&a;
int *u = NULL;
printf("%d %d\n",
a, *t); c = b;
u = t;
printf("%d %d\n", c, *u);
a = 8;
b = 8;
printf("%d %d %d %d\n", b, c, *t, *u);
```

DO THIS: Copy the previous code into your file and test it. Then, add more lines of code to do the following:

1. Update t to point to c. Use a pointer dereference to change the value of c to 555. Verify that it worked by adding a printout. Does this change any of the other values?

2. Change the value of c again using a direct assignment. Verify that the pointer t still points to the value by printing the result of dereferencing it.

Question: Would happen if you tried to execute the following code? How could you fix it? `int *v = &t;`

```
printf("%d\n", *v);
```

This illustrates an important concept: pointers can point to almost anything, even other pointers!

Submission:

Please submit all the answers properly in a report for Task 1.

Task # 02:

Develop a simple task management system using a singly linked list. Each task has a description, a priority level, and a due date. The system should allow users to add, remove, and display tasks, as well as prioritize them based on their due dates.

Define a Task structure with the following fields:

- description: A string describing the task.
- priority: An integer representing the task's priority (e.g., 1 for high, 2 for medium, 3 for low).
- due_date: A string representing the due date (e.g., "2024-08-15").
- next: A pointer to the next task in the list

- **Add Task**: Implement a function to add a new task to the list.

The new task should be inserted based on its due date, with the earliest due date appearing first

- **Remove Task**: Implement a function to remove a task from the list. The function should remove the first occurrence of a task with a given description.

- **Display Tasks**: Implement a function to display all tasks in the list in order, showing their description, priority, and due date.

- **Update Task**: Implement a function to update the priority or due date of an existing task based on its description.

Task # 03:

In this lab, you will work directly with C pointers and structs to manipulate linked lists. The focus of this exercise is to deepen your understanding of pointers, list traversal, and recursive functions by completing and extending an existing C program. You will not use the Scheme-like functions (car, cdr, cons) from the previous lab, but instead, you will rely solely on pointer manipulation to achieve the desired outcomes.

You are provided with a program (namelist.c) that maintains a list of names through a menu-driven interface. This program includes basic functionality to insert, delete, and print names, as well as several unimplemented stubs for additional features. Your task is to complete these stubs and enhance the program by adding new functionality.

- **Initial Exploration:**
  Copy the provided namelist.c file to your account, compile it, and run it several times to understand its current functionality.
  Identify how names are inserted, deleted, and printed from the list.
- **Count the Items in the List:**
  Implement the function countList(const node_t *first) that traverses the list and prints the total number of items.
  The function should iterate through the list node-by-node, counting each item.
- **Print the Last Item in the List:**
  Implement the function printLast(const node_t *first) that prints the data of the last item in the list.
  If the list is empty, the function should print an appropriate message.
- **Pointer Manipulation Review:**
  Before proceeding further, carefully review the provided code. Notice the different types of parameters (node_t * and node_t **) used in various functions like print, printLast, countList, addName, and deleteName.
  Write a brief explanation on why these different types are used and how they impact the functions' behaviors.
- **Move an Item to the Front:**

Implement the function putFirst(node_t **firstPtr) that moves a specified item from anywhere in the list to the front.

This function should:

Search the list for the specified item.

Reassign the necessary pointers to move this item to the front of the list without creating or destroying any nodes.

Handle edge cases, such as when the list is empty or the item is not found.

- **Recursive List Printing:**

  Implement the function printRec(const node_t *first) to print all the elements of the list using recursion instead of iteration.

- **Print the Last Item Recursively:**

  Implement the function printLastRec(const node_t *first) to recursively print the last item in the list.

- **Print the List in Reverse Order**:

  Implement the function printReverse(const node_t *first) to print the names in reverse order, from the last node to the first.

Testing:

Test your program thoroughly with various cases, including:

- An empty list (null).
- A list with one item.
- A list with multiple items, ensuring that all functions behave correctly.

Task # 04:

Your task is to write a C program that takes user input to create two unsorted linked lists of integers. The program should then merge these two linked lists into a single linked list and sort the resulting linked list in ascending order.

- You are provided with two linked lists. The user will enter the number of elements in each list and also enter their values.
- The linked lists are initialized and provided in the form of nodes, where each node contains an integer value and a pointer to the next node in the list.
- Your program should merge the two linked lists into a single linked list and then sort the merged list in ascending order.
- Implement the solution using linked lists. Do not use arrays or other data structures to hold the elements during the merge and sort process.
- Write a function Node* merge_and_sort_lists(Node* list1, Node* list2) that takes the heads of the two linked lists as input and returns the head of the merged and sorted linked list.

Given the following two linked lists:

- List 1: 7 -> 3 -> 5
- List 2: 2 -> 4 -> 1

Your function should return to a merged and sorted list:

- Merged and Sorted List: 1 -> 2 -> 3 -> 4 -> 5 -> 7

Submission:

- The completed namelist.c file with your implementations for task 3.
- A brief written explanation of what you learnt along with test cases and outputs in a report.