CSI 333 – Lecture 5
Introduction to C: Part IV (Pointers)
Pointers

- A variable of type `pointer` contains an `address`.
- Pointers are used heavily in C because
  - They are needed to achieve call-by-reference.
  - Use of pointers leads to compact code (to be seen later).

**Recall:**

- If `x` is a variable, then `&x` gives the address of `x`.
- Think of `&` as an operator that gives the address of its operand.
- In order to do anything with addresses, we need pointer type variables.
Example:

```c
int x, y;
int *p1, *p2;
```

After the above declaration:

- Variable `p1` is of type “`int *`” or “pointer to `int`”.
- Variable `*p1` is of type `int`.
- Similar comments apply to `p2` and `*p2`.

Important Note:

- The above declarations allocate space **only** for the pointer variables.
- They **don’t** allocate space for the corresponding integers.
Since \&x is also of type \texttt{int *}, the following statement is valid:

\[
p1 = \&x;
\]

**Effect:**

The pointer \texttt{p1} “points to” \texttt{x}.

**Notes:**

- The operation \texttt{*p1} \texttt{dereferences} the pointer \texttt{p1}.
- Now, \texttt{*p1} and \texttt{x} refer to the same variable.
After the assignment

\[ p1 = \&x; \]

the following code segment

\[ x = 4; \ printf("%d\n", \*p1); \]

outputs the value 4.

Likewise, the code segment

\[ (*p1) = 7; \ printf("%d\n", x); \]

sets \( x \) to 7 and outputs the value 7.

Further,

\[ (*p1)++; \ printf("%d\n", x); \]

increments \( x \) to 8 and outputs the value 8.
If \( y \) is another int variable, the code segment
\[
y = (*p1) + 5; \quad \text{printf("\%d\n", y);}\]
sets \( y \) to 13 (and prints the value 13).

\section*{Watch out!!}

The declaration
\[
\text{int } *p, q;
\]
declares \( p \) to be of type “int *” and \( q \) to be of type int (and not “int *”).

\section*{Note:}
The special pointer constant NULL denotes a pointer that doesn’t point to anything.
Facility for a program to allocate and deallocate memory during execution.

Important for creating and maintaining data structures such as linked lists, trees, etc.

Dynamic storage allocation and deallocation are done through the C standard library functions malloc and free.

**Code segment:** See Handout 5.1.
Key Points about malloc

■ **Ref:** Deitel & Deitel: Section 12.3.

■ Similar to the `new` operator in Java and C++.

■ The input to `malloc` is the number of bytes to be allocated. (Using "sizeof" enhances portability of code.)

■ Function `malloc`:
  ■ Returns address of the first byte of allocated memory if the allocation was successful.
  ■ Returns `NULL` if allocation failed.
  ■ Type of return value is "`void *`" (the generic pointer type in C).
The returned pointer must be typecast into the appropriate pointer type ("int *" in Handout 5.1) before it is used.

**Important Note**

You MUST always check if the value returned by `malloc` is NULL.

**Variants of malloc:** To be seen later.
Key Points about `free`

- Similar to the `delete` operator in Java or C++.
- The parameter passed to `free` is a pointer to memory previously allocated by a call to `malloc` (or its variants).
- The call to `free` deallocates the memory pointed to by the specified parameter.
- The return type of `free` is `void`. 
A Common Error: Segmentation Fault

Example:

```c
#include <stdio.h>
int main(void) {
    int *p; p = NULL;
    printf("Value = %d\n", *p); return 0;
} /* End of main. */
```

- The program does not have any syntax errors.
- When executed, the above program terminates with a segmentation fault.
- **Reason:** The printf statement dereferences the pointer `p` whose value is `NULL`; that is, no storage has been allocated for the integer to which `p` can point.
- On many systems, segmentation fault also occurs when an uninitialized pointer is dereferenced.
**Assumption:** Each node of the list has an integer and a pointer to the next node.

**Struct and variable declaration:**

```c
struct node {
    int value; struct node *next;
};
struct node *head, *temp;
```

- In the struct definition, “struct node *” represents the corresponding pointer type.
- Variables head and temp are both of type “struct node *”; that is, they are both pointer variables.
- An empty list can be created using the statement
  ```c
  head = NULL;
  ```
Allocating Space for a New Node

**Code Segment:**

```c
temp = (struct node *) malloc(sizeof(struct node));
if (temp == NULL) {
    printf("Allocation failed.\n"); exit(1);
}
```

- It is common to combine the call to `malloc` and the comparison with `NULL`. (See Handout 5.1.)

- Note how the `sizeof` operator is used; even if the definition of `struct node` is changed, there is no need to change the call to `malloc`.

**Note:** The library `<stdlib.h>` is needed for the `exit` function.
Accessing the Fields of a Node

**Method I:** (Not common)

\[(\ast \text{temp}).\text{value} = 47; \ (\ast \text{temp}).\text{next} = \text{NULL};\]

**Method II:** (Commonly used)

\[\text{temp}->\text{value} = 47; \ \text{temp}->\text{next} = \text{NULL};\]

**Program Examples 8 and 9:** See Handouts 5.2 and 5.3.

**Notes on Program Example 9:**

- The `insert_node` function uses arguments of type `struct node **` (i.e., a pointer to a pointer).
- This is needed since we may need to change the values of `head` and/or `tail` pointers when a node is inserted into a list.
Important Note:

Pointers are **not** integers. So, the assignment in the following code segment is **not** allowed:

```c
int *x; x = 12400;
```

**Usefulness of typedef:**

```c
typedef struct node {
    int value; struct node *next;
} NODE, *PTRNODE;
```

- The `typedef` statement allows us to introduce new type names.
With the above typedef:

- The new type name `NODE` represents `struct node`.
- The new type name `PTRNODE` represents `struct node *`.

Now, we can have the following declarations:

```
NODE x;  PTRNODE head, tail;
```

The new type name can be used wherever the longer version of the type name is used (e.g. as the return type for functions).
Suggested Exercises

1. Write a function

   ```c
   struct node *last_occur (struct node *h, int x)
   ```

   which returns a pointer to the last occurrence of a node which contains the value given by the parameter \( x \) in the list pointed to by \( h \). The function should return \text{NULL} if the value given by \( x \) does not occur in the list.

2. Write a function

   ```c
   void delete_node (struct node **ph, int x)
   ```

   which deletes the first node that contains the value given by \( x \) in the list pointed to by \( *ph \). If the list does not contain a node with value given by \( x \), the function should leave the list unchanged.