Part VI
Pointers

1) Introduction
2) Declaring Pointer Variables
3) Using Pointers
4) Pointer Arithmetic
5) Pointers and Arrays
6) Pointers and Structures
7) Pointers and Functions
8) Common Pointer Pitfalls
9) Complex Example
6.1 Introduction

- **Pointer:**
  - Scalar data type
  - Stores an address of memory

- **Idea:**

- **Examples:**
  - pointer to a structured variable

- **Components:**
  - Pointer name
  - Type of item pointed to

- **C uses pointers explicitly with:**
  - Arrays
  - Structures
  - Functions
6.2 Declaring Pointer Variables

\[
\text{<type> '*' <pointer_name>';}'
\]

- **Notes:**
  - Declares the variable `<pointer_name`
  - Is a pointer to a memory location of type `<type`
  - Allocates memory for the pointer
  - Does not initialise the pointer

- **Components:**
  - Pointer name:
    \[
    \text{<pointer_name> = <name>}
    \]
  - Pointed type:
    \[
    \text{<type>}
    \]

- **Usage:**

```c
typedef struct personal_tag
{
    int birth_year;
    char name [20];
    int ID;
    char school [4];
} personal_type;

personal_type* ptr1;
int* ptr2;
float* ptr3;
```

**Note:** ptr2 is a pointer to an integer
### 6.3 Using Pointers

- **Example:**

  ```
  int a;
  int* ptr_a;
  float b;
  float* ptr_b;
  a = 100;
  b = 500.0;
  ptr_a = &a;
  ptr_b = &b;
  printf("%d", &a);
  printf("%d", ptr_a);
  printf("%f", b);
  printf("%f", *ptr_b);
  printf("%d", a);
  printf("%d", *ptr_a);
  printf("%5.2f", b);
  printf("%5.2f", *ptr_b);
  ```

- **Note:**
  - `&` operator gives the address of a variable
  - `*` operator gives the content of an object pointed to by a pointer
6.4 Pointer Arithmetic

- **Note:**
  - The addition and subtraction involving pointers are performed taking into account the size of the type the pointer points to.

  - Assuming the declarations:
    ```
    <type>* ptr;
    int n;
    ```

  - The expression:
    ```
    ptr + n
    ```

  - Is evaluated as follows:
    ```
    ptr + n * (sizeof(<type>)
    ```

- **Example:**
  ```
  int* ptr_a;
  int a;
  int n;

  ptr_a = &a;   //let assume ptr_a = 20120
  n = 5;

  ptr_a = ptr_a + n; // ptr_a = 20120 + 20
  ```
6.5 Pointers and Arrays

• **Observation:**
  - All the items of an array have the same size equal to the size of their type
  - Array elements are arranged in consecutive memory locations

  int marks[3]; \( \Rightarrow 4B \times 3 = 12\) B

• **Consequence:**
  - Pointers can be used to access array items

• **Example:**

  ```c
  int i;
  int* ptr;  \[1\]
  int marks[3];  \[2\]

  for (i = 0; i < 3; i++)  \[3\]
  {
    scanf(“%d”, &marks[i]);
  }
  ```
ptr = marks; // ptr = &marks[0];  

for (i = 0; i < 3; i++)  
{
    printf("%d", *(ptr + i));
}

Effect:

Note:
ptr = marks same as ptr = &marks[0]
*(ptr+2) and marks[2] access the same object!
6.6 Pointers and Structures

- **Observation:**
  - Computer allocates a contiguous memory block for storing a structure
  - The allocated memory space is equal to the added size of all the structure’s items

```c
typedef struct coord_tag struct coord_tag
{
    float x;
    float y;
    float z;
    char text[3]
} coord_type;
```

- **Static Memory Space Allocation**

  ```c
  coord_type point  OR  struct coord_tag point
  
  \[4B \times 3 + 1B \times 3 = 15 B\]
  ```

  ![Memory Allocation Diagram](image-url)
• Pointer Declaration

coord_type* ptr    OR    struct coord_tag* ptr

• Pointer Usage

ptr = &point;    //  ptr = 200
ptr->x = 1.0;    //  (*ptr).x = 1.0;
ptr->y = 2.0;
ptr->z = 0.0;    //  scanf("%f", &(ptr->z));
strcpy(ptr->text, "Mon");
6.7 Pointers and Functions

- **Observation:**
  - Function arguments can be passed in two ways
    - **BY-VALUE** or
    - **BY-ADDRESS** => in the function, the value of the argument can be modified and the new value is maintained after the function has finished

- Pointers provide the solution: *Pass the address of variables to the function*

```c
int a, b;
my_func(&a, &b);
```

- **Example ("by-address")**
  Write a function that swaps the values of the two arguments

```c
void swap(int* px, int* py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

```c
int a, b;
a = 2;
b = 5;
swap(&a, &b);
```
6.8 Common Pointer Pitfalls

- **Uninitialised Pointers**

Not assigning a pointer to a memory address before using it

```c
int *x;
*x = 100;
```

**WRONG!**

‘x’ is not initialised in the program with a physical location (address)

C compiler initialises a pointer (or variable) with a random value when a pointer (or variable) is declared

=> X points to a random mem. location. We cannot update information from that location

**THE RIGHT WAY**

```c
int* x;
int y = 20;
x=&y; //pointer is initialised with an address
*x = 100; //y will be equal with 100
```

- **Illegal indirection**

```c
char* p;
*p = (char*)malloc(100);
*p = 'y';
```

**WRONG!**

Function `malloc()` allocates memory dynamically (at run time)

`malloc()` returns a `pointer` to the allocated block of memory if successful or a NULL pointer otherwise.
THE RIGHT WAY

\[ p = (\text{char}*)\text{malloc}(100); \]

If code rectified as above, one problem is if no memory
is available and \( p \) is NULL.

Therefore we cannot assign:
\[ *p = 'y'; \]

A good C program would check for this:
\[
\begin{align*}
   p &= (\text{char}*)\text{malloc}(100); \\
   \text{if} \ (p == \text{NULL}) &\{ \\
   \quad \text{printf(“Error: Out of Memory!!”);} \\
   \quad \text{exit(1);} \\
   \} \\
   \end{align*}
\]

\[ *p = 'y'; \]

- **Omitted ‘*’**

If you want the value of the location to which a pointer \( p \)
points, you need to use the expression \( *p \) and not \( p \)

\[
\begin{align*}
   \text{int* } x; \\
   \text{int } y = 20; \\
   x &= &y; \\
   \text{printf("Here is the value: %d", x);} \\
\end{align*}
\]

**WRONG!**
You are asking C compiler to print the
value contained in the pointer \( x \), which
is an address of memory, rather than
the value contained at that address.
THE RIGHT WAY

```c
int* x;
int y = 20;
x = &y;
printf("Here is the value: %d",*x);
```

- Omitted ‘&’

It is just as easy to forget an ‘&’ as it is to forget a ‘*’

**Example 1:**

```c
int x = 0;
printf("No: ");
scanf("%d", x);
```

**WRONG!**

- scanf() receives as parameter a pointer (memory address).
- By forgetting the &, the value of x ( = 0) is passed instead.
- scanf() stores the number that it read from the keyboard into memory location 0 => this is wrong!

THE RIGHT WAY

```c
int x = 0;
printf("No: ");
scanf("%d", &x);
```
Example 2:

```c
void my_func (int* n);
int x = 0;
my_func(x);
```

**WRONG!**

We are supposed to pass a pointer as an argument to my_func(), and we gave it an integer instead.

**THE RIGHT WAY**

```c
void my_func (int* n);
int x = 0;
my_func(&x);
```

- **Multiple Pointers Declaration**

  ```c
  char* c1, c2;
  ```

  **WRONG!**

  This does not declare c2 to be char *, but just char.

  c1 is a pointer.

  **THE RIGHT WAY**

  ```c
  char *c1, *c2;
  ```

  **Note:** char* c1; behaves like char *c1;

- **Memory Allocation**

  ```c
  char *str;
  strcpy(str, "Harry");
  ```

  **WRONG!**

  It is not enough just to declare the pointer for a mem. location where a string will be stored

  Storage space must be allocated for the actual string, before storing any information
THE RIGHT WAY

char *str;
str = (char*)malloc(10);
strcpy(str, "Harry");

• Operator Precedence

What is the value of z?
x = 7; y=14;  z = x + y * 2 +1 =>
z = 7 + 14 * 2 + 1 =>  z = 43 ?  z = 49 ? z = 36 ?

Convention: Multiplication takes precedence over addition
=> Correct: z = 36

C Operator Precedence Table

E.g. of operators: ( ) [ ] . ++ -- + - = (Assignment)
 * (dereference) * (multiplication)

In order to save remembering the C precedence rules, it is strongly suggested that you always use parentheses to indicate the correct order in which to perform operations.

Some examples: the dangers of remembering wrong the precedence rules

Example 1: Copy a string from src to dest

void mystrcpy(int* dest, int* src, int size)
{
    for (i=0; i<size; i++)
        *dest++ = *src++;
}

Expression *p++ increments p or what it points to?
**Answer:** *p++ increments p

*p++ is equal with *(src++)

++ and * have equal precedence, but they associate right-to-left.

**Ex:** if src points to the first element then *src++ first returns 1 and then the increment takes effect and the pointer is stepped one element along the array.

**HOW** we increment what p points to?

**Answer:** (*p)++

**Ex:** if src points to the first element then (*src)++ returns value 1 that will be incremented => value 2.

What ++*src would do?

for (i=0; i<size; i++)

*dest++ = ++*src;

**Answer:** ++*src increments the value pointed by src

++*src is equal with ++$(*src)

**NOTE!**

Make sure that you understand how pointers work and especially how * and & are used with them!
Note:

- * means indirection (dereference),
- ++ or -- mean increment; either pre- or post-increment
- the combinations can be pre- or post-increment of either the pointer or the data it points to, depending on where the brackets are put

<table>
<thead>
<tr>
<th>++(*p)</th>
<th>pre-increment data pointed to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*p)++</td>
<td>post-increment data pointed to</td>
</tr>
<tr>
<td>*(p++)</td>
<td>access data via pointer, post-increment pointer</td>
</tr>
<tr>
<td>*(++p)</td>
<td>access via pointer which has already been incremented</td>
</tr>
</tbody>
</table>

Example 2

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

void main()
{
    int x, *p;
    p = &x;   /* initialise pointer */
    *p = 0;   /* set x to zero */
    printf("x is %d\n", x);
    printf("*p is %d\n", *p);

    *p += 1;   /* increment what p points to */
    printf("x is %d\n", x);

    (*p)++;    /* increment what p points to */
    printf("x is %d\n", x);
    exit(EXIT_SUCCESS);
}
```

Output:  X is 0
         *p is 0
         x is 1
         x is 2
         *p += 1 same as *p = *p + 1
         (*p)++ same as *p = *p + 1
6.9. Complex Example

```c
int main()
{
    int i, j, k;
    int *p, *q, *r;
    int **s, **t, **u;

    i = 7;
    j = 23;
    k = i + j;

    p = &k;
    q = &i;
    r = p;

    *r = k + 1;
    *p = (*p + *r) * 5;
    // (31+31)*5
    i = *(q);

    s = &q;
    u = &p;
    t = s;
}
```
```c
p = *t;

*p = **u * i;
// i=i*i

**s = i + j + k;
//i=i+j+k

j = 0;

r = p = &j;

s = &r;
t = s;

u = &p;
p = &k;

**u *= (**p * **s) * (i * j * k);
//k=k*(k*j)*(i*j*k)

printf("i = %d\n", i); //i = 862
printf("j = %d\n", j); //j = 0
printf("k = %d\n", k); //k = 0
return (EXIT_SUCCESS);
```