Review of C - Pointers & Arrays

Prepared by Vibha Masti

References

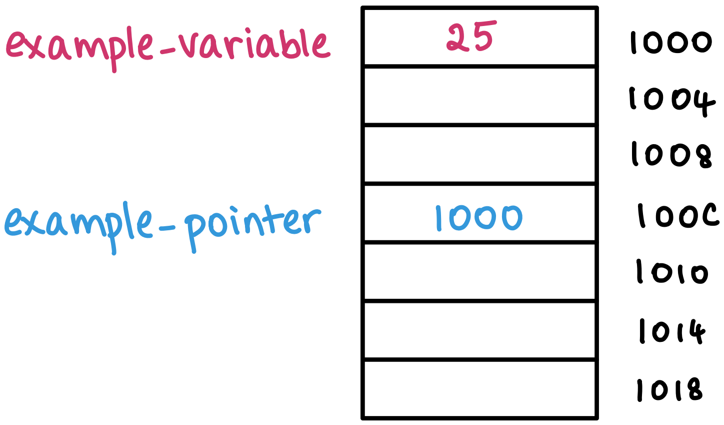
- "The C Programming Language. 2nd Edition", Brian Kernighan and Dennis Ritchie
- Geeks for Geeeks

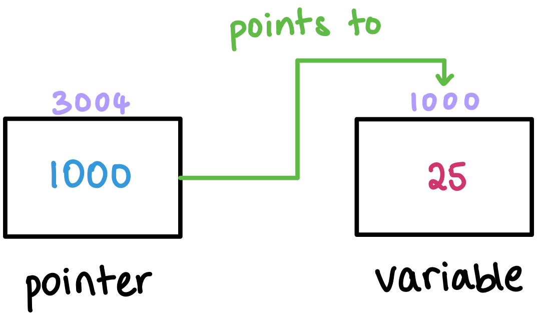
Acknowledgement

- Dr. Shylaja Sharath

Pointers

- A pointer is a variable that stores the address of another variable
- In C, the **unary** operator `&` is used to find the address of a variable
- A pointer is defined using the `*` operator
- To **dereference** a pointer (i.e., access the object that the pointer is pointing to), the unary dereferencing operator `*` is used





```c
/* An int variable */
int example_variable = 25;

/* Pointer to the variable */
int *example_pointer = &example_variable;

/* Dereferencing the pointer - modifies the original variable */
*example_pointer = 35;
```

- The declaration below indicates that the expression `*ip` is an `int`

```c
int *ip;
```

Pointer types

- Pointers are constrained to point to variables of a particular **type** (the exception being `void` pointers)
- For instance, if the pointer `ip` points to an integer `a`, then the expression `*ip` can legally occur wherever the expression `a` occurs

```c
int age = 25;
int *ip = &age;

printf("In ten years, you will be %d years old.\n", age + 10);

printf("In twenty years, you will be %d years old.\n", *ip + 20);
```

- An `int` pointer cannot point to a variable of type `float`, or any other type (compiler generates a warning and it is a dangerous practice)

Safe practice: initialise all declared pointers to `NULL`

```c
int *ip = NULL; // safe
int *ip;        // not safe
```

Pointers as Arguments

- Pointers can be passed as function arguments
- C passes function arguments only by value, so passing pointers enables functions to alter values of variables in the calling function

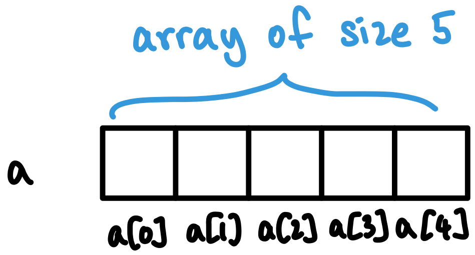
Example program: swapping

```
void swap(int *px, int *py) {
  /* Swaps values of *px and *py */
    int temp = *px;
    *px = *py;
    *py = temp;
}

int main() {
    int x = 20, y = 30;
    /* Before: x = 20, y = 30 */
    swap(&x, &y);
    /* After: x = 30, y = 20 */
    return 0;
}
```

Pointers and Arrays

- An array is a collection of items of the same type stored at contiguous memory locations
- Any element in the array is accessed by offsetting from the array's base address
- Arrays in C are zero-indexed



- Any operation that can be achieved with array subscripting can be achieved with pointers
- The name of the array is the pointer to the first element of the array (base address)

```
/* Declare an array of size 5 */
int *a[5];

/* Indexing arrays - access 3rd element */
printf("%d\n", a[2]);

/* Using pointers - access 3rd element */
printf("%d\n", *(a + 2))
```

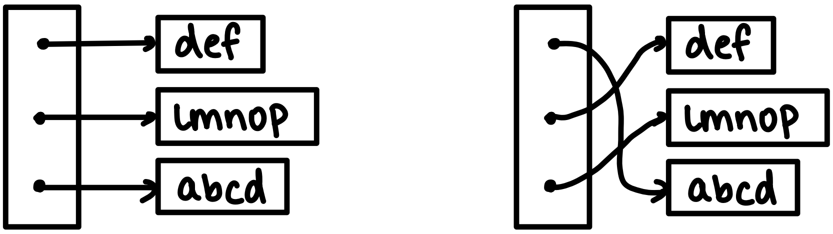
- The same applies to character arrays and pointers

```
char *a = "Hello World!";

char b[] = "Hello World!";
```

Pointer Arrays (Pointers to Pointers)

- A common use case for an array of pointers is an array of character strings
- Sorting an array of character strings invoves just rearranging the pointers in the array, and not copying the strings themselves into new memory locations



Methods to define an array of strings

Method 1

```
// An array of size 4 with elements of type char*
char *names[4] = {"James", "Betty", "Eliza", "Bella"};

// Print all names
for (int i = 0; i < 4; ++i) {
    printf("%s\n", names[i]);
}
```

Method 2

```
// An array of size implicitly defined with elements of type char*
char *names[] = {"James", "Betty", "Eliza", "Bella"};

// Print all names
for (int i = 0; i < 4; ++i) {
    printf("%s\n", names[i]);
}
```

Method 3

```
// A two-dimentional array with number of columns = 5
char names[][5] = {"James", "Betty", "Eliza", "Bella"};

// Print all names
for (int i = 0; i < 4; ++i) {
    printf("%s\n", names[i]);
}
```

Not allowed - generates a segmentation fault

```
// No memory allocated for all the strings
char **names = {"James", "Betty", "Eliza", "Bella"};

for (int i = 0; i < 4; ++i) {
    printf("%s\n", names[i]);
}
```

Can individually assign to indices after allocating memory

```
char **names = NULL;

names = (char **) malloc(5*4*sizeof(char));

names[0] = "James";
names[1] = "Betty";
names[2] = "Eliza";
names[3] = "Bella";

for (int i = 0; i < 4; ++i) {
    printf("%s\n", names[i]);
}
```

Double Pointers

- A double pointer is a pointer to a pointer
- Dereferencing a double pointer once returns a single pointer

```
int a = 25;     /* variable */

int *ip = &a;   /* pointer*/
int **dp = &ip; /* double pointer */

/* Dereferencing the double pointer once */
if (ip == *dp) {
    print("This is true\n");
}

/* Dereferencing the double pointer twice */
if (a == **dp) {
    print("This is true\n");
}

/* Dereferencing the pointer once */
if (a == *ip) {
    print("This is true\n");
}
```

Pointer to array vs Array of Pointers

- A pointer to an array is a pointer than stores the address containing the first element of the array, but it is not of type `int *`
- A pointer to an array points to the whole array and not just the first element
- Incrementing a pointer to an array increments it by the size of the array and not by the size of an `int`

```
/*Array of 5 integers */
int arr[5] = {1, 2, 3, 4, 5};

/* Pointer to an array of 5 integers */
int (*parr)[5] = arr;

/* Pointer to the first element of the array */
int *p = arr;
```

- An array of pointers is an array containing pointers as its elements

```
/* Array of 10 integer pointers */
int *p[10];
```

Multi-dimensional Arrays and Pointers

- A multi-dimensional array can be declared in C as shown below

```
/* 5x5 matrix */
int m[5][5] = {
  {1, 2, 6, 4, 5},
  {2, 2, 3, 1, 4},
  {3, 6, 2, 2, 6},
  {4, 7, 9, 4, 1},
  {5, 8, 0, 2, 3}
};

/* 5x5 matrix (implicit number of rows) */
int m[][5] = {
  {1, 2, 6, 4, 5},
  {2, 2, 3, 1, 4},
  {3, 6, 2, 2, 6},
  {4, 7, 9, 4, 1},
  {5, 8, 0, 2, 3}
};
```

- Must specify number of columns

```
/* Not allowed - must specify number of columns */
int m[][] = {
  {1, 2, 6, 4, 5},
  {2, 2, 3, 1, 4},
  {3, 6, 2, 2, 6},
  {4, 7, 9, 4, 1},
  {5, 8, 0, 2, 3}
};
```

- Accessing the i^{th} row and j^{th} column of a multi-dimensional array can be done using indexing or pointers

```
/* Indexing */
int x = m[i][j];

/* Pointers */
int y = *(*(m + i) + j)
```