Pointers in C

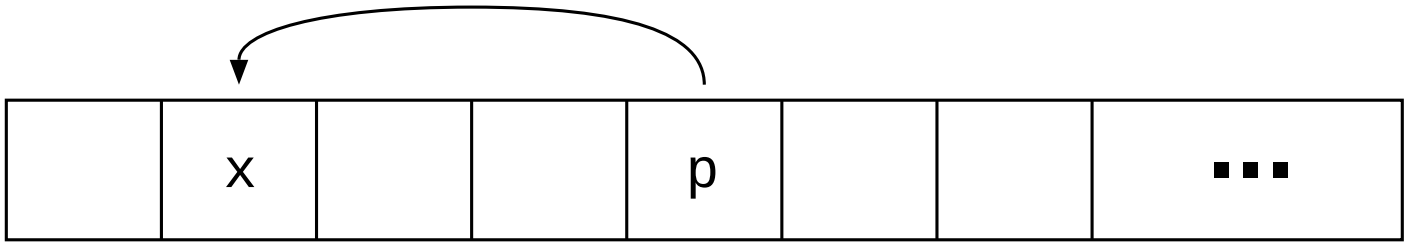
Computing Lab

https://www.isical.ac.in/~dfslab

Indian Statistical Institute

Pointers

- Memory = consecutively numbered storage cells (*bytes*)
- Variable can occupy one or more contiguous bytes, depending on its type
- *Address* of a variable = serial number of “first” byte occupied by the variable
- *Pointer* holds the address of a variable
- Pointer / address itself may be (usually is) stored in another variable



Basic operations

`&` – address / location operator

`*` – *dereferencing* operator

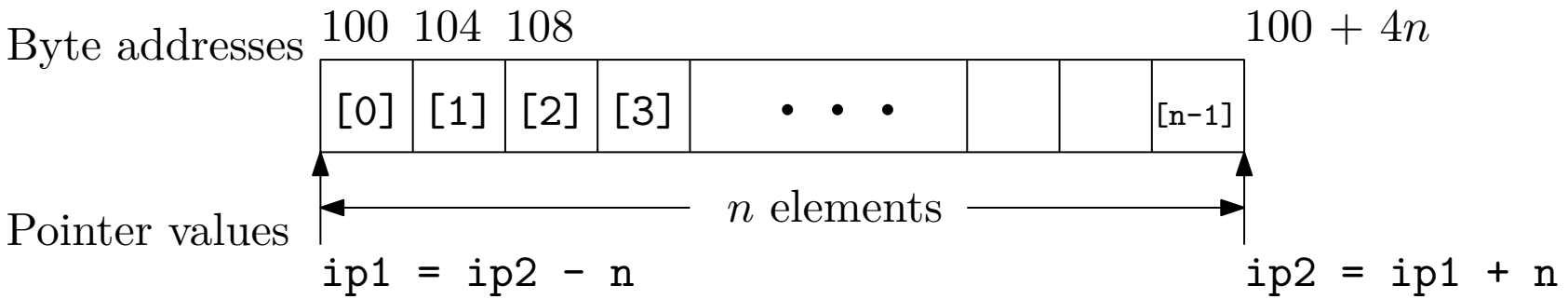
```
char c, *cp; /* neither c nor cp is initialised */
int i, *ip;  /* neither i nor ip is initialised */

cp = &c;  ip = &i; /* cp, ip are initialised now */
*cp = 0; /* same as c = 0;  c is initialised now */
```

Example

```
char str0[8] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
char str1[8] = { 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q' };
char str2[8] = { 'z', 'y', 'x', 'w', 'v', 'u', 't', 's' };
char *cp;

cp = &str0[0];
printf("%p %p %c\n", cp, &cp, *cp);
printf("%p %p %p\n", &str0[0], &str1[0], &str2[0]);
printf("%p %p %p\n", &str0[7], &str1[7], &str2[7]);
```

```
$ ./a.out
0x7fff5f56e870 0x7fff5f56e868 a
0x7fff5f56e870 0x7fff5f56e878 0x7fff5f56e880
0x7fff5f56e877 0x7fff5f56e87f 0x7fff5f56e887
```

Pointer arithmetic



`ip1 + n`	points to n-th *element* (of the proper type) after what `ip` is pointing to
`ip2 - n`	points to n-th *element* (of the proper type) before what `ip` is pointing to
`ip2 - ip1`	number of elements between `ip1` and `ip2`

Pointers and arrays

An array name is synonymous with the address of its first element.

Conversely, a pointer can be regarded as an array of elements starting from wherever it is pointing.

```
int a[10] = {...}, *p;

p = a;        /* same as p = &(a[0]); */
*p = 5;       /* same as a[0] = 5; */
p[2] = 6;     /* same as a[2] = 6; */
*(a+3) = 7;   /* same as a[3] = 7; */
```

But:

CORRECT	INCORRECT
`&p`	`&a`
`p = a;`	`a = p;`
`p++;`	`a++;`

Pointer-array equivalence (contd.)

Using pointer arithmetic	Using array elements
`p = a + i`	`p = &(a[i])`
`*p = x`	`a[i] = x`
`*(p+j) = x`	`p[j] = x` or `a[i+j] = x`

Review questions

1. What does the following code do and why? (see `strcpy.c`)

```
1  char a[32] = "Introduction", b[32] = "Programming", *s, *t;
2  s = a; t = b;
3  while (*s++ = *t++);
```

2. What output is generated by the following code and why?

```
for (i=0; i < 10; i++)
    printf("abcdefghijklmnop\n" + i);
```

Review questions — Solutions

1. String copying

```
do {
  *s = *t;
  s++; t++;
} while (*t != '\0');
```

Review questions — Solutions

1. String copying

```
do {
  *s = *t;
  s++; t++;
} while (*t != '\0');
```

```
do {
  *s++ = *t++;
} while (*t != '\0');
```

Review questions — Solutions

1. String copying

```
do {
  *s = *t;
  s++; t++;
} while (*t != '\0');
```

```
do {
  *s++ = *t++;
} while (*t != '\0');
```

```
while ((*s++ = *t++) != '\0');
```

Review questions — Solutions

2. Think of the problem this way:

```
p = "abcdefghijklmnop\n";
printf(p);
```

Review questions — Solutions

2. Think of the problem this way:

```
p = "abcdefghijklmnop\n";
printf(p);
```

```
p = "abcdefghijklmnop\n";
printf(p + 2);
```

Review questions — Solutions

2. Think of the problem this way:

```
p = "abcdefghijklmnop\n";
printf(p);
```

```
p = "abcdefghijklmnop\n";
printf(p + 2);
```

```
p = "abcdefghijklmnop\n";
printf(p + i);
```

Topics to be covered

1. How do you allocate space for an array if you do not know (a reasonable upper bound on) the size when writing your program?
2. What to do if an array is full, and you need to store more elements?
3. Multi-dimensional arrays
4. Difference between `int a[M][N]` and `int **a;` ⟵ LATER

Variable length arrays (VLAs)

OK

```
int num_elts;

scanf("%d", &num_elts);

int array[num_elts];
```

WRONG

```
int num_elts;           // not initialised
int array[num_elts];    // num_elts == ???
```

Caution: (more detailed explanation later)

- Local variables allocated on stack
- Maximum stack size limited (often 8 MiB)
- Large local VLAs may not work

  Example: compile and run `large-vlas.c`; experiment with the array sizes in the program.

Alternative: use global / static / dynamic allocation

VLAs (contd.)

Reference: `https://en.cppreference.com/w/c/language/array`

- Expression evaluated + array allocated each time flow of control passes over the declaration
- Expression's value must be positive
- Array should not be accessed after declaration goes out of scope

  Exercise: is it actually deallocated?
- Cannot be members of structs / unions

Allocating memory

Syntax:

```
#include <stdlib.h>
(type *) malloc(n * sizeof(type))
(type *) calloc(n, sizeof(type))
(type *) realloc(ptr, n * sizeof(type))

free(ptr)
```

`malloc`, `calloc`, `realloc` return `void` pointers

Convenient macros: (see `common.h`)

```
#define Malloc(n,type) (type *) malloc( (unsigned) ((n)*sizeof(type)))
#define Realloc(loc,n,type) (type *) realloc( (char *)(loc), \
                            (unsigned) ((n)*sizeof(type)))
```

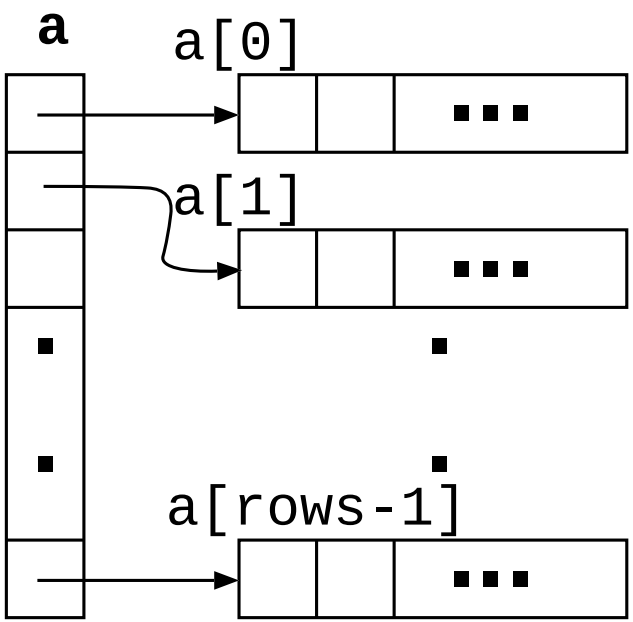
Extending an array using `realloc`

```
int *array, capacity = 100, num_elts = 0;

/* Initial allocation */
if (NULL == (array = Malloc(capacity, int))) {
    perror("out of memory");
    exit(1);  // instead of exit(0)
}

...

/* "Grow" the array when required */
if (num_elts == capacity) {
    capacity *= 2;
    if (NULL == (array = Realloc(array, capacity, int)) {
        perror("out of memory");
        exit(1);
    }
}
```

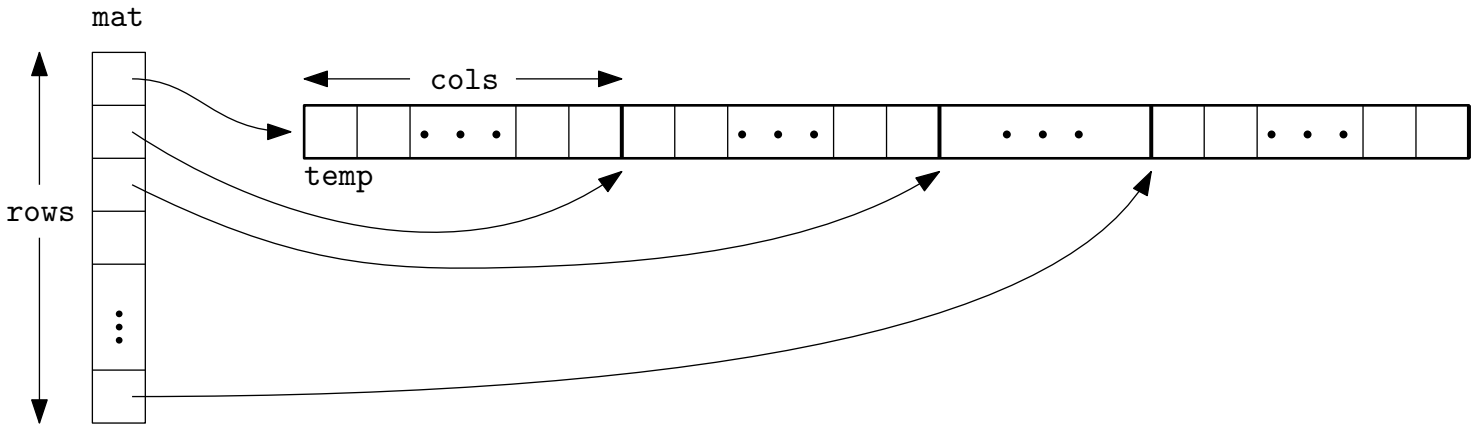
Multi-dimensional arrays

Multi-dimensional array = array of arrays = pointer to pointer

```
int **a, i;
a = (int **) malloc(rows * sizeof(int *));
for (i = 0; i < rows; i++)
    a[i] = (int *) malloc(cols * sizeof(int));
```



Multi-dimensional arrays: row-major storage




```
int ii;
int *temp;
if (NULL == (temp = (int *) malloc(rows*cols*sizeof(int))) ||
    NULL == (mat = (int **) malloc(rows * sizeof(int *))))
    ERR_MESG("Out of memory");
for (ii = 0; ii < rows; temp += cols, ii++)
    mat[ii] = temp;
```

Programming problems

1. Consider 2 sequences of letters (a–z), A and B, stored in arrays.
   (a) Write a program to find the number of (possibly overlapping) occurrences of the sequence B in A.
   (b) Write a program to find whether the multisets corresponding to A and B are equal.
2. Write a program that first reads multiple lines of text from the terminal, and then, depending on the user's choice, prints either the odd- or the even-numbered lines, either in their original or in reverse order.

   You may assume that
   - lines are numbered starting with one;
   - each line is no more than 80 characters long;
   - the input text will not consist of more than 10 lines.

   Redesign your program so that it will run correctly even if the number of lines in the input text is not known a priori.