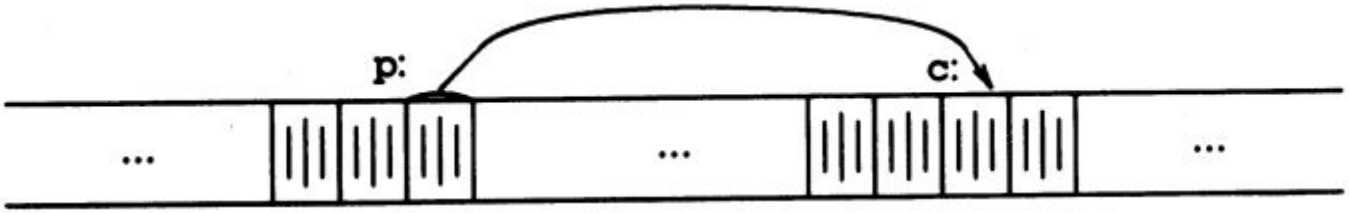
Introduction to C - p2

Instructor: Vinicius Prado da Fonseca, PhD (vpradodafons@online.mun.ca)

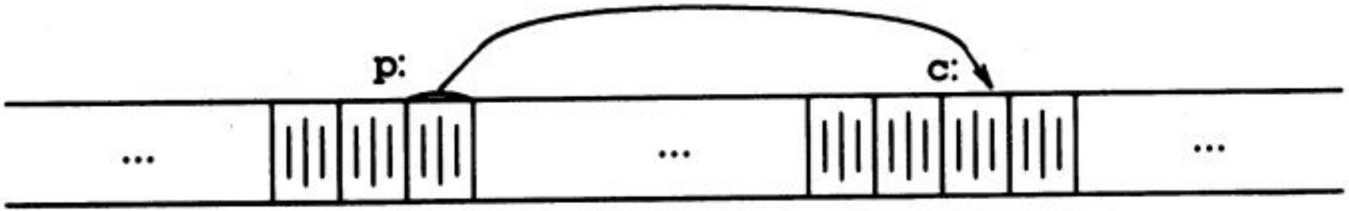
5.1 Pointers

- Memory is an consecutive numbered memory cells
- Pointer is a group of those cells that holds an address
- Operator & gives the address of something p = &c;
- Operator * access the object that pointer is pointing to



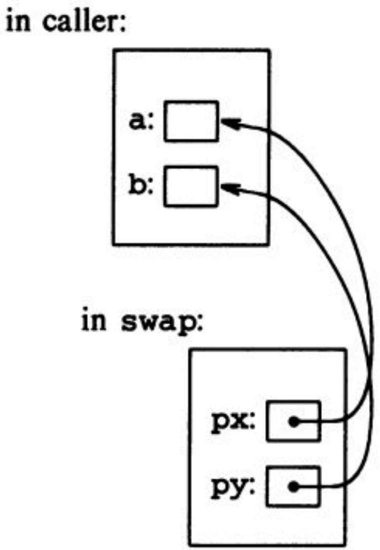
5.1 Pointers

```
int x = 1, y = 2;
int *ip;            // It reads "pointer to an integer"
                    // The expression "*ip" is an int
ip = &x;            // ip receives the address of x
y = *ip;            // copy the content of whatever ip points to y
*ip = *ip + 10;     // *ip affects x
```



5.2 Pointers and function arguments

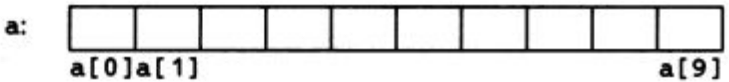
- C passes arguments to functions by value
- No direct way to alter the variable in the calling function (scope)



```
int swap(int x, int y) {
    int temp;

    temp = x;
    x = y;
    y = temp;
}

int pswap(int *px, int *py) {
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

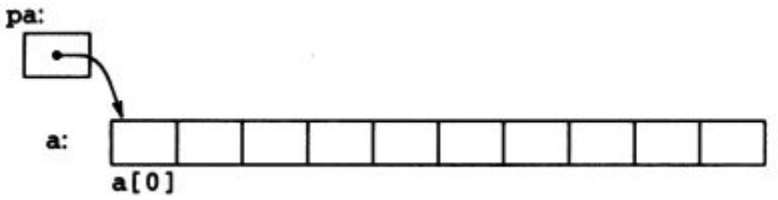
5.3 Pointers and arrays

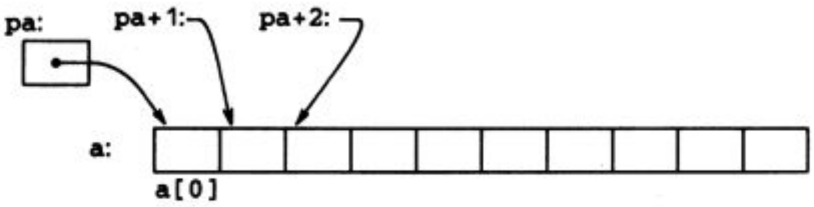
- Strong relationship between pointers and arrays
- Any operation that can be achieved indexing (arr[i]) can also be done with pointers;
- In general it will be faster with pointers



5.3 Pointers and arrays

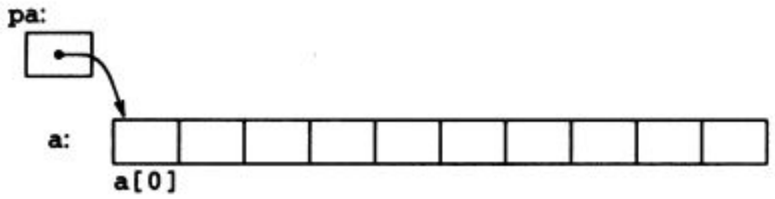
- Strong relationship between pointers and arrays
- Any operation that can be achieved indexing (arr[i]) can also be done with pointers;
- In general it will be faster with pointers

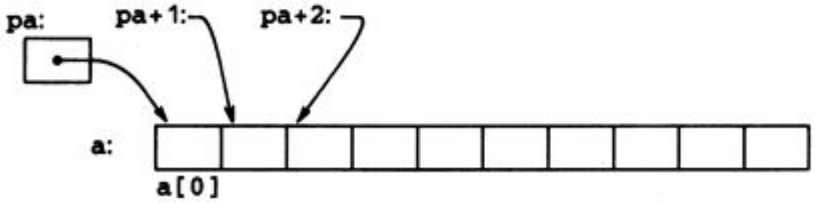




5.3 Pointers and arrays

- Compilation time allocation (e.g. char line[MAXLINE])
    - Scope allocation/deallocation
- Static allocation, fixed size
    - Ex: Truncate smaller string with “\0”
- Request memory size during run time
- Dynamic allocation moving pointers

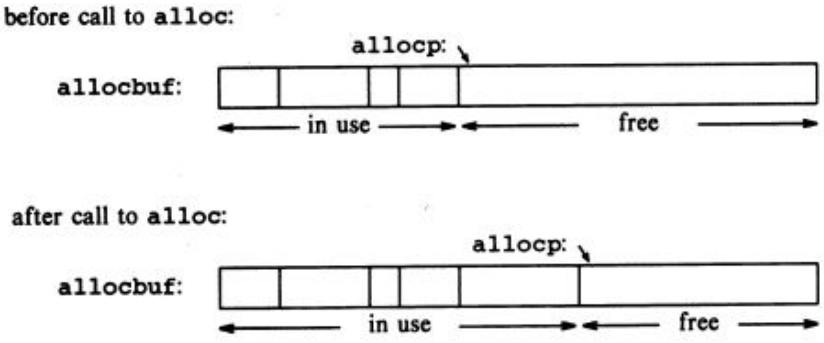




5.4 Address Arithmetic

Runtime allocation

- Stack-like allocation
- char allocbuff[ALLOCSIZE]
- char *allocp = allocbuf
- alloc(n)
    - Pointer p = allocp
        - allocp can be only visible by alloc and free
    - Move allocp requested size n
        - allocp + n positions
    - Return a pointer p
- afree(p)
    - Move the allocp to position p
    - Stack, last in first out.
- Correct call order



5.4 Address Arithmetic

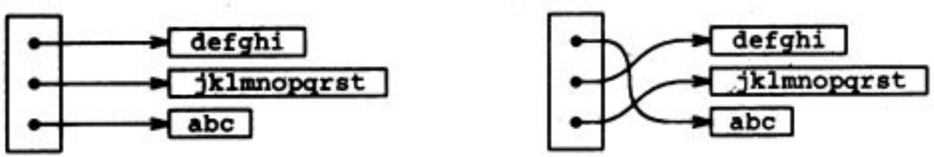
Runtime allocation

- allocbuff[ALLOCSIZE]
- alloc(n)
    - Pointer p = allocp
        - allocbuff and allocp can be only visible by alloc and free
    - Move allocp requested size n
        - allocp + n positions
    - Return a pointer p
- afree(p)
    - Move the allocp to position p
    - Stack, last in first out.
- Call alloc/afree correct order

```
char *alloc(int n) {

  // will it fit?

  if (allocbuf + ALLOCSIZE - allocp >= n) {

    allocp += n;

    return allocp - n;

  } else

    return 0;

}
```

5.6 Array of Pointers; Pointers to Pointers

- Pointers are variables themselves
  - Can be stored in arrays, like other variables
- Points to the first letter of the text
- Sorting is faster, just swap pointers
  - 5.6pointerarrays_swap.c
- Complete sort example in the book



5.9 Pointers vs Multi-dimensional arrays

- Multi-dimensional arrays

  int a[10][20];

  200 int-sized locations

- Pointers

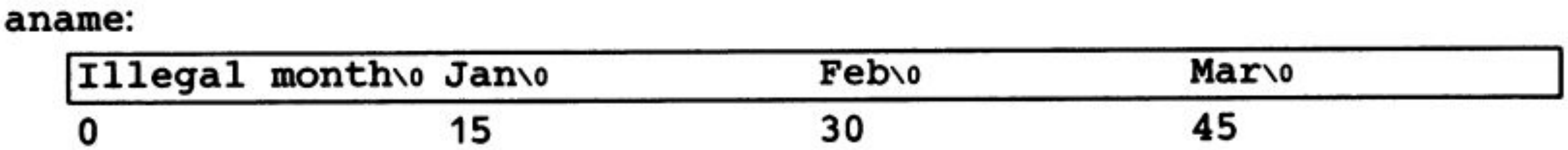
  int *b[10]

  Using 20-element arrays

  200 ints + 10 pointers

- Advantage is rows may have different lengths

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```



5.9 Pointers vs Multi-dimensional arrays

- Multi-dimensional arrays

    int a[10][20];

    200 int-sized locations

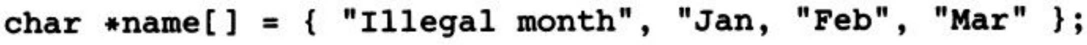
- Pointers

    int *b[10]

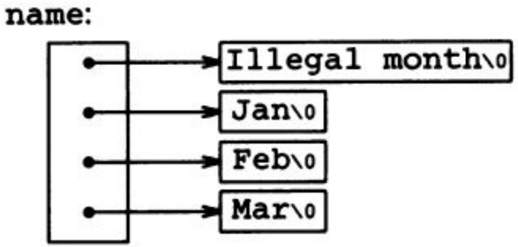
    Using 20-element arrays

    200 ints + 10 pointers

- Advantage is rows may have different lengths



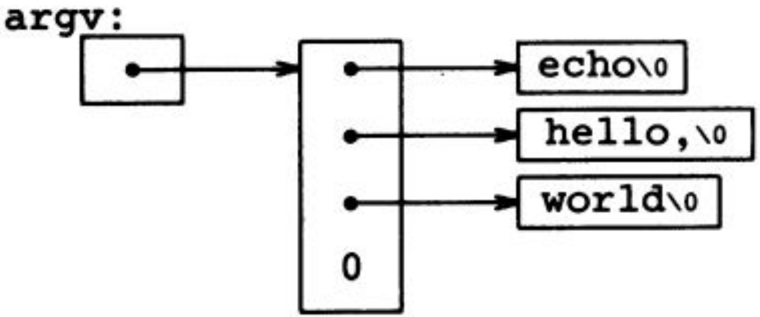
```
char *name[] = { "Illegal month", "Jan, "Feb", "Mar" };
```



5.10 Command-line Arguments

./echo hello, world

- argc
  - Argument counter
- argv
  - Argument vector



6.1 Basic of Structures

- Collection of one or more variables
  - Possibly different types
  - Grouped together
  - Treated as a unit
- Ex. Employee record
  - Name
  - Age
  - Job title
- typedef

```
struct point {

        int x;

        int y;

};
```

6.2 Structures and Functions

- Manipulate structures
- Pass components separately
- Pass the entire structure
- Pass a pointer

```
struct point init_pt(int x, int y) {
    struct point temp_pt;
    temp_pt.x = x;
    temp_pt.y = y;


    return temp_pt;
}
```

6.2 Structures and Functions

- Manipulate structures
- Pass components separately
- Pass the entire structure
- Pass a pointer

```
struct point init_pt(int x, int y) {
    struct point temp_pt;
    temp_pt.x = x;
    temp_pt.y = y;


    return temp_pt;
}
```

point-list.c

- Structure with a pointer to the next node

  Point.next = previous

- typedef
- pointnode is a tag
  - make the struct not unknown, reference after
- But typedef define PointNode as a type, like int. More common.

```
typedef struct pointnode {

        struct pointnode *nextpoint;

        int x;

        int y;

} PointNode;
```

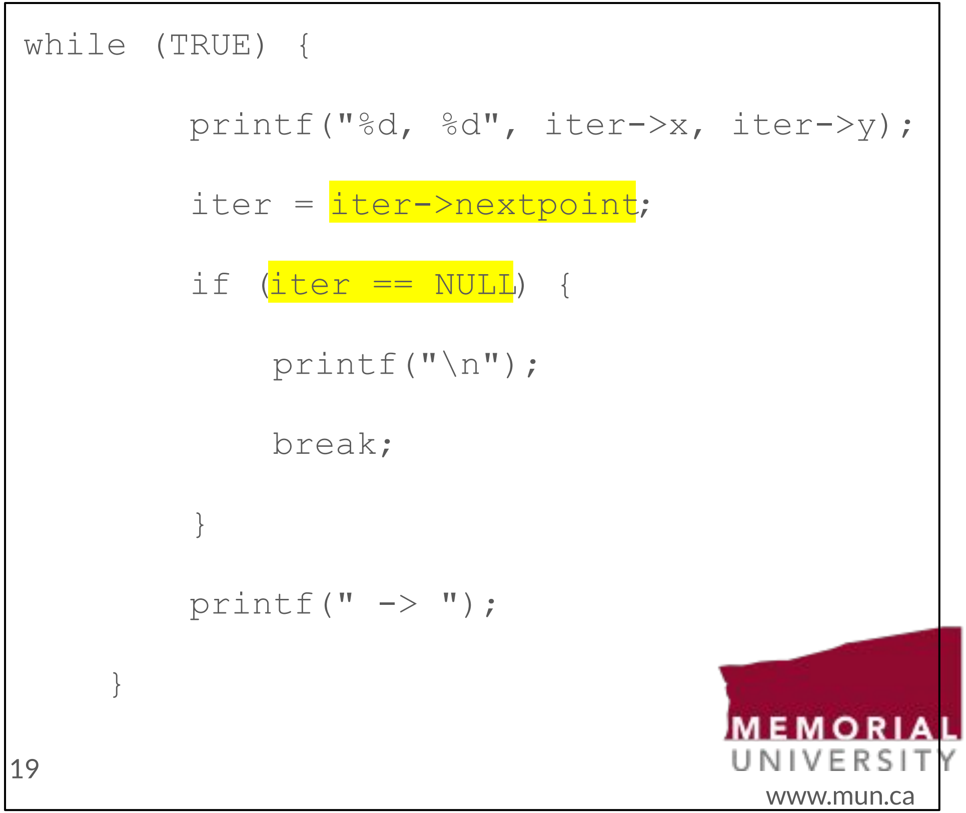
point-list.c

- Dynamic allocation
    - malloc
- sizeof(PointNode)
    - Size in bytes
- (PointNode *)
    - Cast for the type we need

```
PointNode *palloc(void) {

          return (PointNode *) malloc(sizeof(PointNode));

}
```

point-list.c

- Printing example
- Follow the nextpoint
- Chained list



```
while (TRUE) {
        printf("%d, %d", iter->x, iter->y);
        iter = iter->nextpoint;
        if (iter == NULL) {
            printf("\n");
            break;
        }
        printf(" -> ");
    }
```

Questions?

Try examples and code yourself

- Really type the examples
- Do not copy and paste

https://github.com/vncprado/intro-to-c