CSC 4630

Supplementary Material on C

Review of Pointers

Checklist on pointers in C (do you remember how these work?)

* and & operators

NULL and undefined values

assigning values to pointers

```
p = &v;
p = malloc(sizeof(int));        /*(<stdlib.h>; (int *) not necessary)    */
```

pointers and arrays - the same but different

```
*(p+i) == p[i]
```

pointer arithmetic

strings and their implementation

pointers as function arguments

```
$ cat pointers.c
/*      pointers.c   (not so intimidating)    */

int     n;
int     *p;

int main(void)
{
        n = 4630;
        p = &n;             /* address of n */

        printf("n is %d\n", n);
        printf("n's address is %ld\n", &n);
        printf("p holds n's address, so its value of course is %ld\n", p);
        printf("dereferencing p gives us %d\n", *p);
        printf("p also has an address, which is %ld\n", &p);

        return 0;
}
$
$
$
$

$ pointers
n is 4630
n's address is 137924
p holds n's address, so its value of course is 137924
dereferencing p gives us 4630
p also has an address, which is 137928
$
$
```

Pointers Overview

- What is a pointer?

  A variable that contains an address

- Why use pointers?

  To efficiently access data

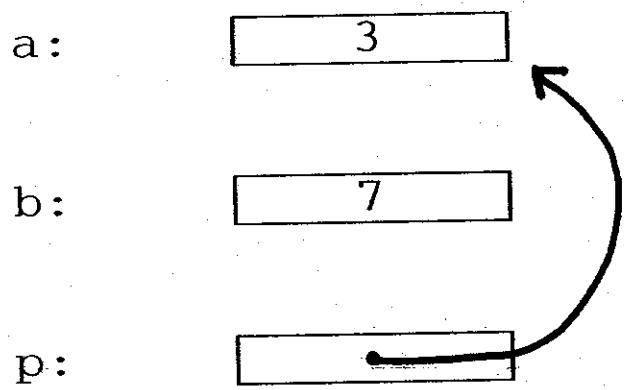
  To write flexible code

  To change variables passed to a function

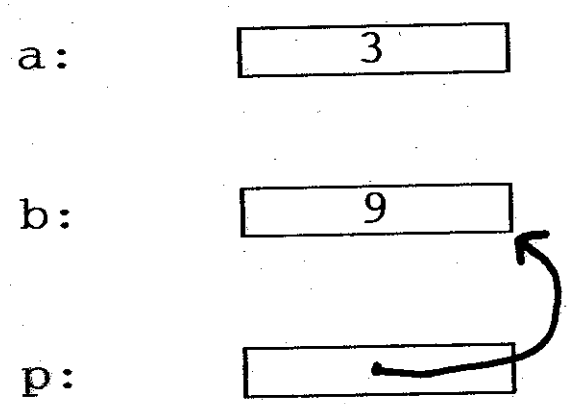
  To work with dynamically allocated memory

  To access hard-coded addresses in system code

```
/* two int's and a pointer to an int */
int    a, b, *p;

a = b = 7;

/* p is assigned the address of a */
p = &a;

/* Object pointed to by p (i.e. a) gets 3 */
*p = 3;
```



```
/* two int's and a pointer to an int */
int    a, b, *p;

a = b = 7;

/* p is assigned the address of a */
p = &a;

/* Object pointed to by p (i.e. a) gets 3 */
*p = 3;

/* Now p is changed to point to b */
p = &b;

/* input integer into address of b */
scanf("%d", p);
```

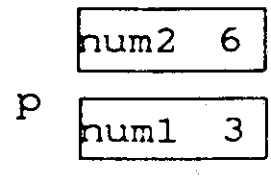


Declaring, Initializing, and Using Pointers

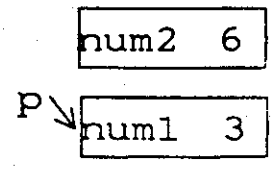
```
#include    <stdio.h>
main()
{
   int  num1=3, num2=6;
   int  *p;

   p = &num1;
   printf("%d\n",*p);

   *p = 20;
   printf("%d ",*p);
   printf("%d\n",num1);

   p = &num2;
   printf("%d\n",*p);
}
```

Hypothetical Stack

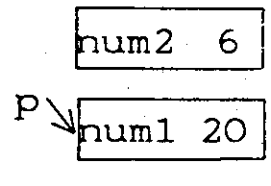
Addr.	*Variable*	*Value*
510	p	?
504	num2	6
500	num1	3



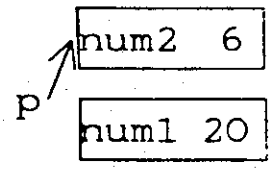
510	p	500
504	num2	6
500	num1	3



510	p	500
504	num2	6
500	num1	20



510	p	504
504	num2	6
500	num1	20



Output:

```
3
20 20
6
```

The Null Pointer

- It is not legal to read from or write to address 0
- A null pointer:

  Pointer with value of 0

  Returned by pointer-returning functions to signify failure

- NULL is defined in stdio.h as

```
#define   NULL   0
```

-or-

```
#define   NULL   (void *) 0
```

POINTERS AND ADDRESSES (Contd)

VARIABLE	DECLARATION	ADDRESS
a	int a;	3000410
b	int b;	3000414
c	int *c;	3000420

INITIALIZE:

```
a = 1;
b = 2;
c = &a;
```

INITIALLY:

&a is 3000410 a is 1	&b is 3000414 b is 2	&c is 3000420 c is 3000410 *c is 1

CHANGE a = 3;

a is 3	b is 2	*c is 3

CHANGE c = &b;

a is 3	b is 2	c is 3000414 *c is 2

Pointer Arithmetic

- Important to declare pointer with correct type

- OK to add to, subtract from, compare pointers and subtract one pointer from another

```
char *p;
```

*p is a character

p++ is equivalent to p = p + (1 byte)

```
int *p;
```

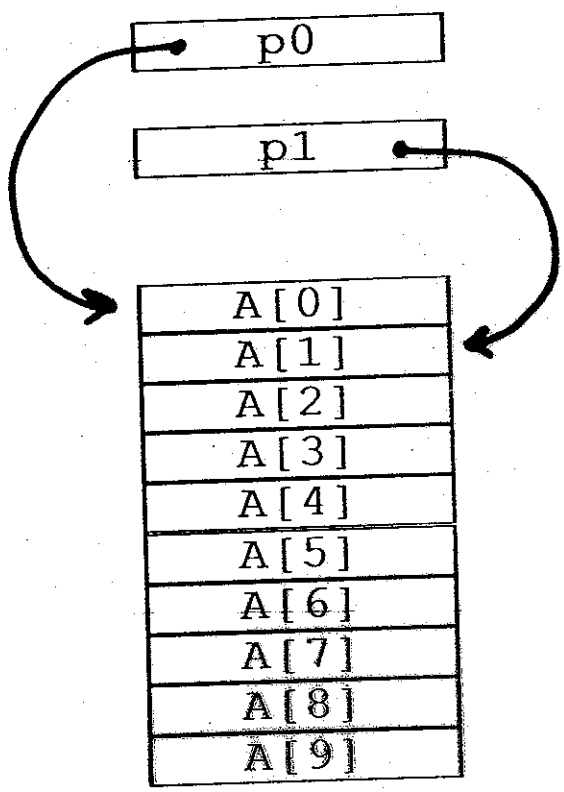
*p is an integer

p++ is equivalent to p = p + (bytes in an int)

POINTER SCALE FACTOR

POINTER TO	SCALE FACTOR
CHARACTER	1
SHORT	2
INTEGER	4
LONG	4
FLOAT	4
DOUBLE	8
ARRAY	SIZE OF ARRAY ELEMENT
STRUCTURE	SIZE OF STRUCTURE (INCLUDING HOLES)

```
/* Declare an array and two pointers */
intA[10], *p0, *p1;

/* Set p0 to point to first element of A */
p0 = A;

/* Set p1 to point to second element of A */
p1 = A + 1;
```



POINTERS VS ARRAYS

- Declaration

```
char   name[35];   /* storage for 35 bytes */

char   *p;         /* storage for 1 address */
```

- Assignment

```
p = name;      /* Pointer initialization */
               /* required before */
               /* p can be used */

name = address        /* Illegal */
```

- The [ ] and * operators may be used with both arrays and pointers.
  - Convention to use

    [ ] with arrays

    ```
    name[i] = 's';
    ```

    * with pointers

    ```
    p = &name[i];
    *p = 's';
    ```

Using a Pointer to Traverse an Array

- For sequential access, * is faster than [ ]

```
 1  /* Reads an input line into an array */
 2  /* Assumes input line is <= LINELEN chars */
 3
 4  #include     <stdio.h>
 5  #define      LINELEN    256
 6
 7  main()
 8  {
 9      char    line[LINELEN + 1], *p;
10
11      p = line;     /* p = &line[0]; */
12      while ((*p = getchar()) != '\n')
13              p++;
14      *p = '\0';
15      printf("%s\n",line);
16  }
```

```
$ a.out
do it    <-- typed by user
do it    <-- output from program
$
```

```
/*        swap happens in the body of main()   */

#include <stdio.h>

int main(void)
{
   int i = 10, j = 20, temp;

   printf("In the beginning, i = %2d, j = %2d\n", i, j);

   temp = i;     /* Here is the "swap"  */
   i = j;
   j = temp;

   printf("At the end, i = %2d, j = %2d\n", i, j);

   return 0;
}
```

```
In the beginning, i = 10, j = 20
At the end, i = 20, j = 10
```

```
/*      swap happens in a function, with only values passed */

#include <stdio.h>

void swap(int n, int m);

int main(void)
{
   int i = 10, j = 20;

   printf("In the beginning, i = %2d, j = %2d\n", i, j);

   swap(i, j);

   printf("At the end, i = %2d, j = %2d\n", i, j);

   return 0;
}

void swap(int x, int y)          /* arguments passed "by value"      */
{
   int temp;

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

   return;
}
```

```
In the beginning, i = 10, j = 20
At the end, i = 10, j = 20
```

```
/*      swap happens in a function, with addresses passed    */

#include <stdio.h>

void swap(int *n, int *m);

int main(void)
{
   int i = 10, j = 20;

   printf("In the beginning, i = %2d, j = %2d\n", i, j);

   swap(&i, &j);

   printf("At the end, i = %2d, j = %2d\n", i, j);

   return 0;
}

void swap(int *x, int *y)                /*  arguments passed as addresses */
{
   int temp;

   temp = *x;
   *x = *y;
   *y = temp;

   return;
}
```

```
In the beginning, i = 10, j = 20
At the end, i = 20, j = 10
```