Arrays and Pointers (part 1)

CSE 2031

Fall 2012

Arrays

- Grouping of data **of the same type.**
- Loops commonly used for manipulation.
- Programmers set array sizes explicitly.

Arrays: Example

- Syntax

```
type name[size];
```

- Examples

```
int bigArray[10];
double a[3];
char grade[10], oneGrade;
```

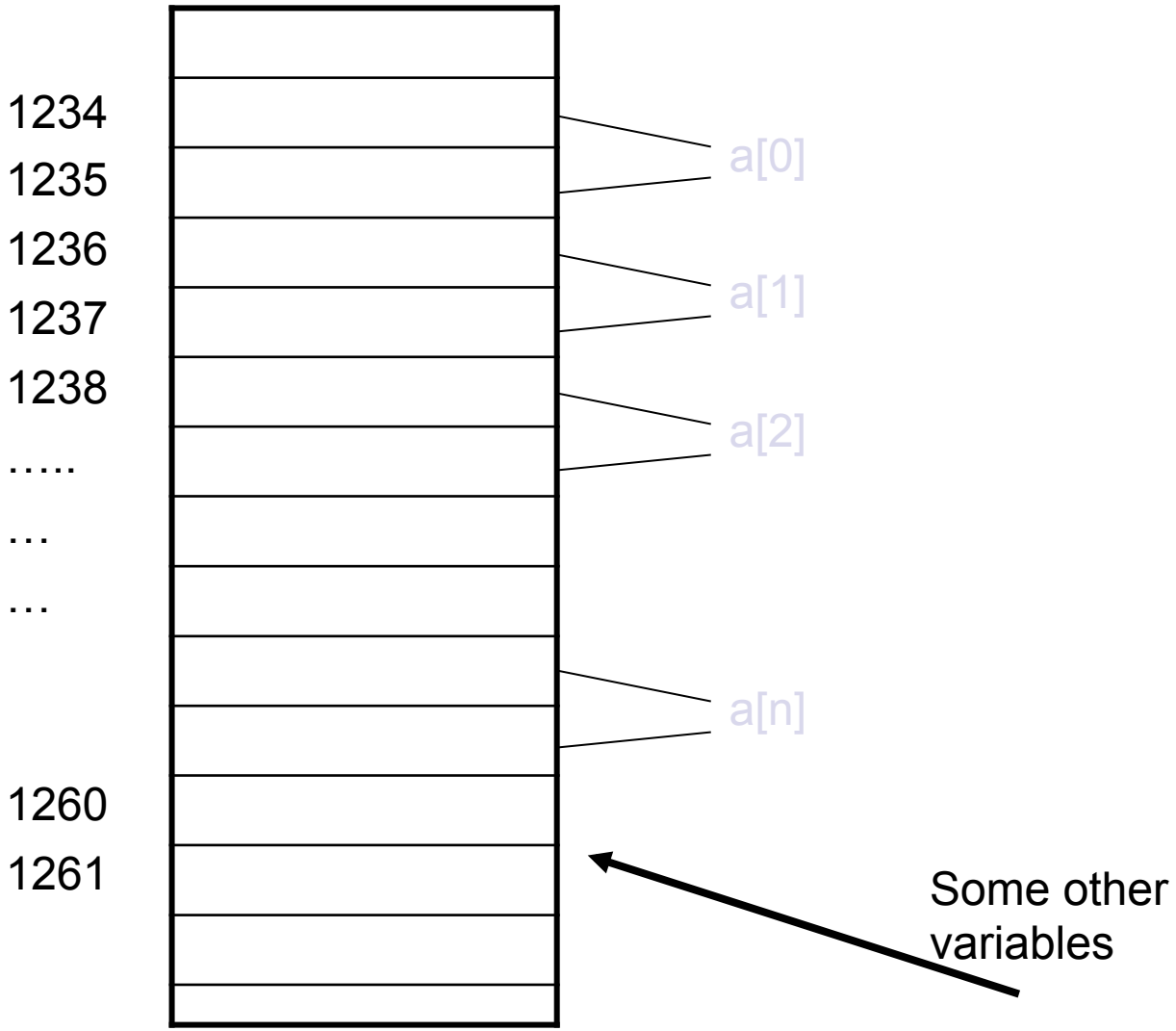
Arrays: Definition and Access

- Defining an array: allocates memory

  ```
  int score[5];
  ```

  - Allocates an array of 5 integers named "score"
- Individual parts can be called:
  - Indexed or subscripted variables
  - "Elements" of the array
- Value in brackets called index or subscript
  - Numbered from 0 to (size – 1)

Arrays Stored in Memory



Initialization

- In declarations enclosed in curly braces

int a[5] = {11,22};

Declares array a and initializes first two elements and all remaining set to zero

int b[ ] = {1,2,8,9,5};

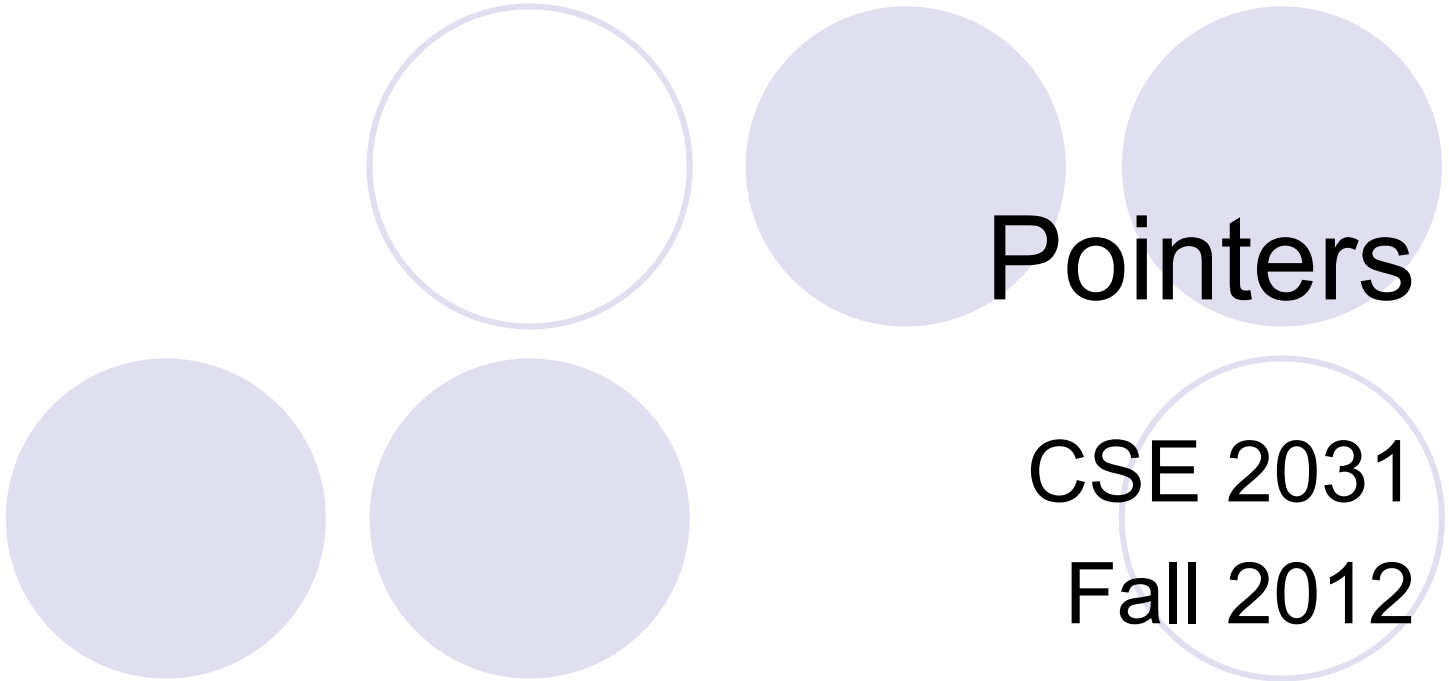
Declares array b and initializes all elements and sets the length of the array to 5

Array Access

```
x = ar[2];
ar[3] = 2.7;
```

- What is the difference between

  `ar[i]++, ar[i++], ar[++i]` ?



Pointers

CSE 2031

Fall 2012

Pointers and Addresses (5.1)

- Memory address of a variable
- Declared with data type, * and identifier
  `type *pointer_var1, *pointer_var2, …;`
- Example.
  ```
  double *p;
  int *p1, *p2;
  ```
- There has to be a **`*`** before EACH of the pointer variables

Pointers and Addresses (cont.)

- Use the **"address of"** operator (&)
- General form:

```
pointer_variable = &ordinary_variable
```


Name of the pointer

Name of ordinary variable

Using a Pointer Variable

- Can be used to access a value
- Unary operator * used
  - * pointer_variable
    - In executable statement, indicates value

- Example

```
int *p1, v1;
v1 = 0;
p1 = &v1;
*p1 = 42;
printf("%d\n",v1);
printf("%d\n,*p1);
```

Output:
42
42

Pointer Example 1

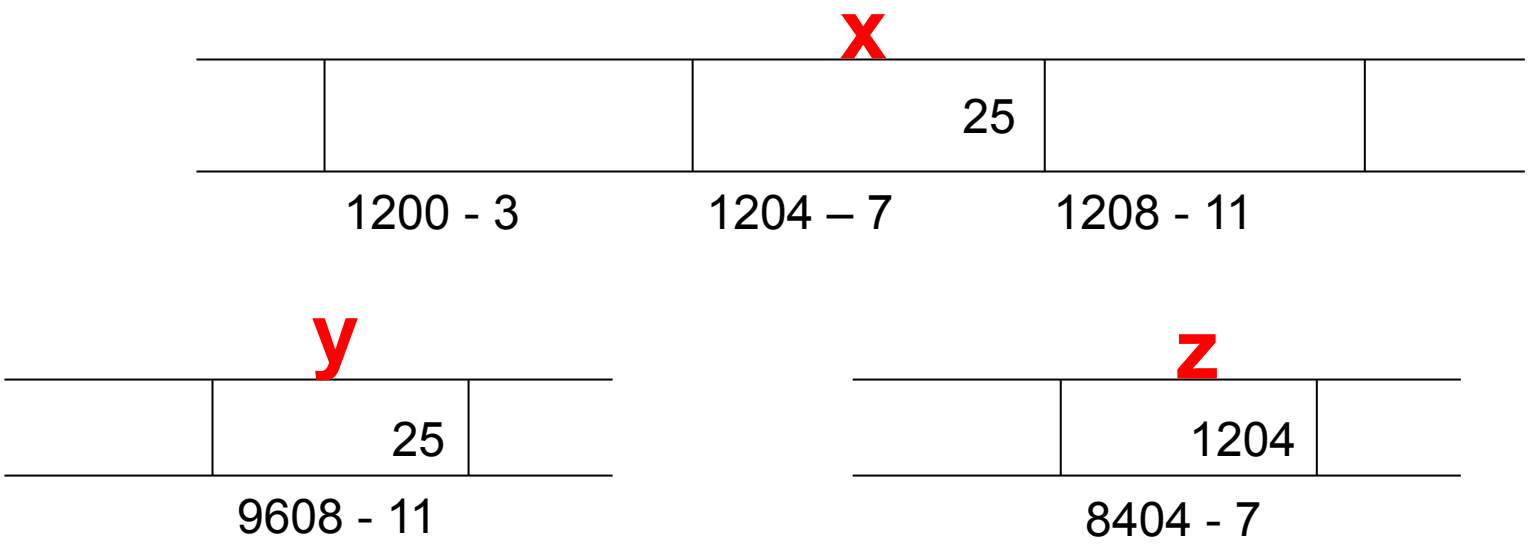
int x,y;

int *z;

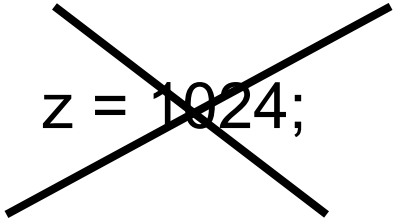
x = 25;

y = x;

z = &x;



Pointer Variables

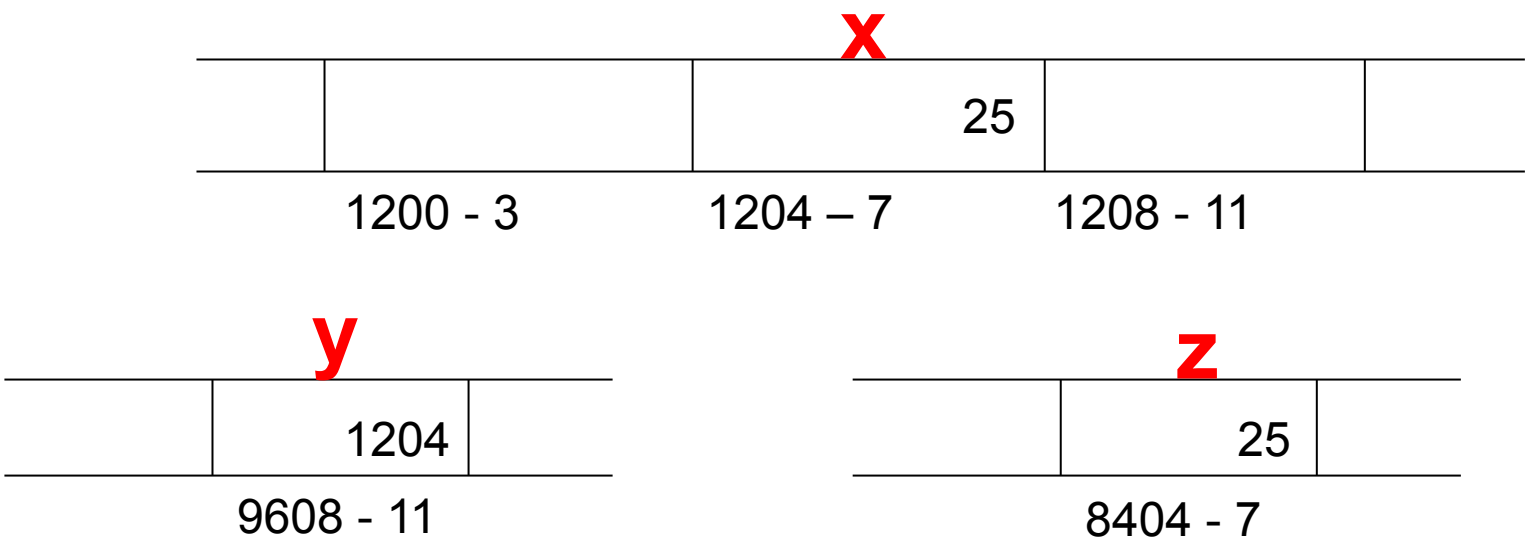


BAD idea

Instead, use z = &x

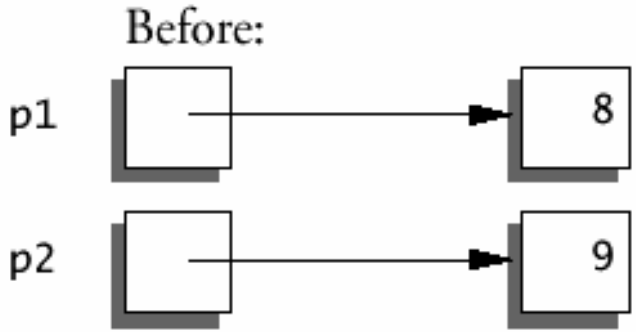
Pointer Example 2

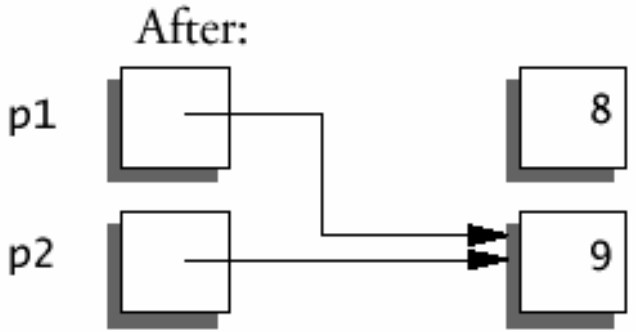
```
int x = 25, *y, z;

y = &x;

z = *y;
```



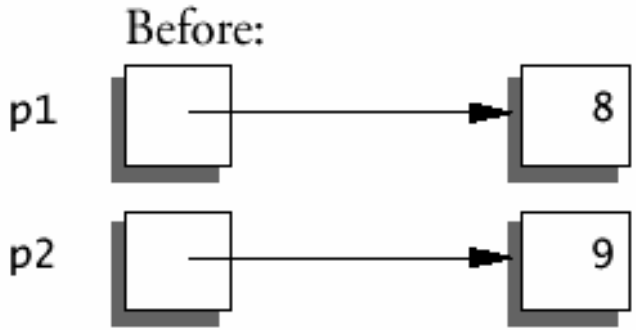
Pointer Example 3

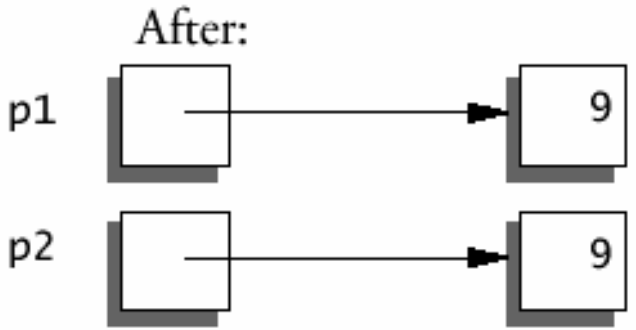
```
p1 = p2;
```






```
*p1 = *p2;
```





More Examples

```
int x = 1, y = 2, z[10], k;
int *ip;
ip = &x;     /* ip points to x*/
y = *ip;     /* y is now 1 */
*ip = 0;     /* x is now 0 */
z[0] = 0;
ip = &z[0]; /* ip points to z[0] */
for (k = 0; k < 10; k++)
  z[k] = *ip + k;
*ip = *ip + 100;
++*ip;
(*ip)++;     /* How about *ip++ ??? */
```

Pointers and Function Arguments (5.2)

Write a function that swaps the contents of two integers a and b.

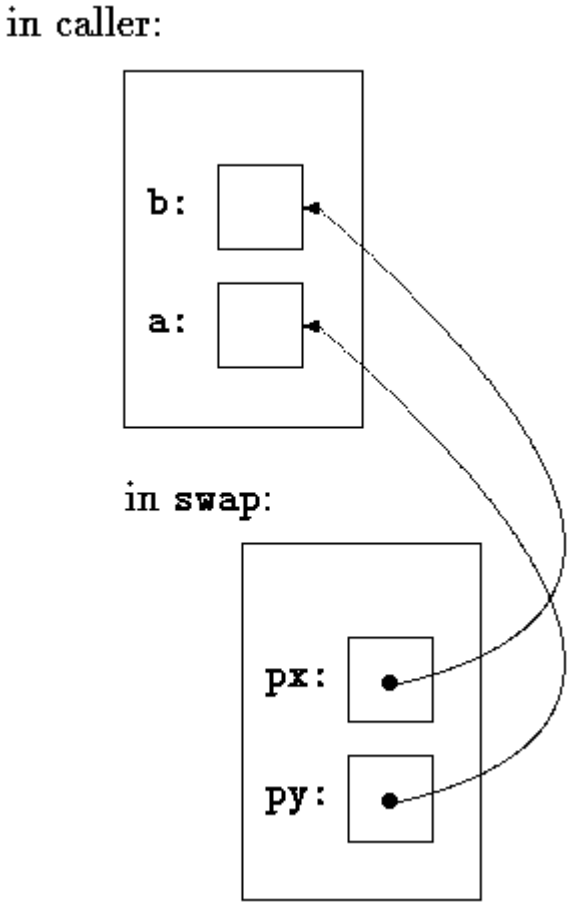
```
void main( ) {
    int a, b;
    /* Input a and b */
    swap(a, b);
    printf("%d %d", a, b);
{
```

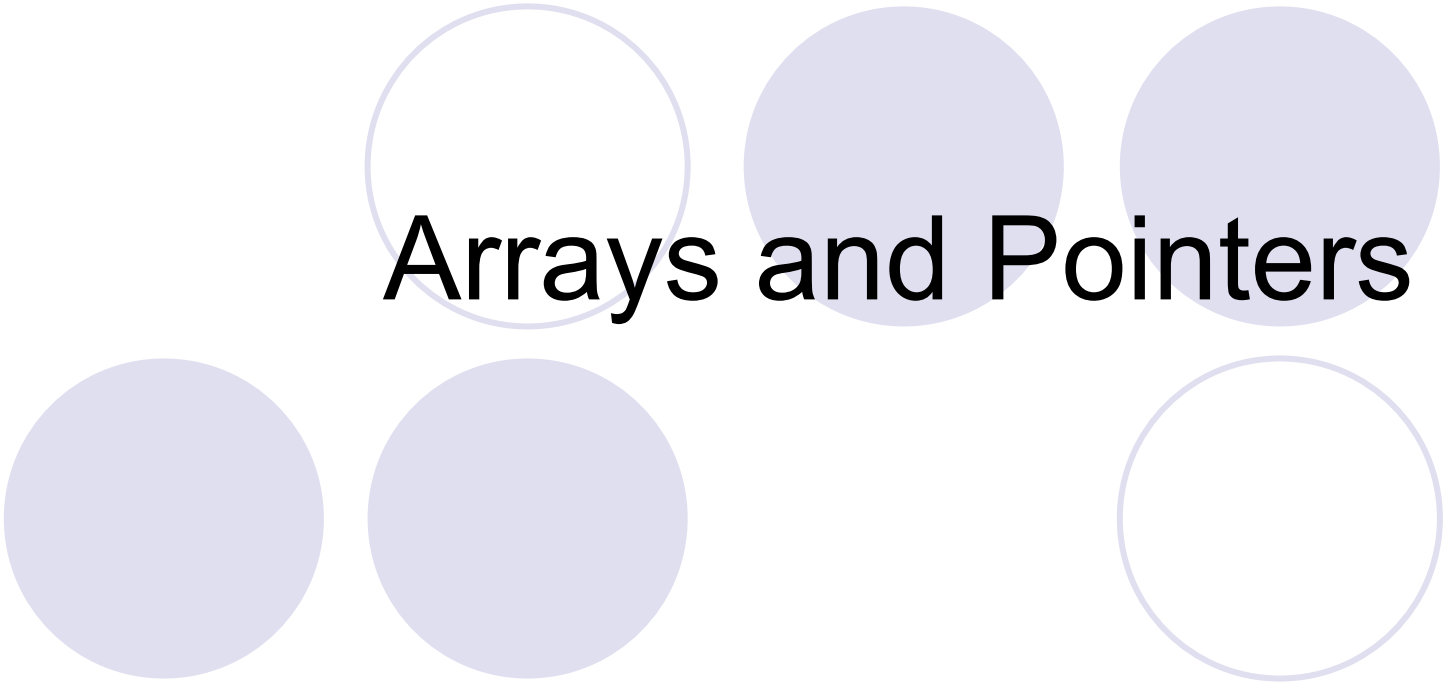
C passes arguments to functions by values.

```
void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

The Correct Version

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

void main( ) {
    int a, b;
    /* Input a and b */
    swap(&a, &b);
    printf("%d %d", a, b);
{
```





Arrays and Pointers

Pointers and Arrays (5.3)

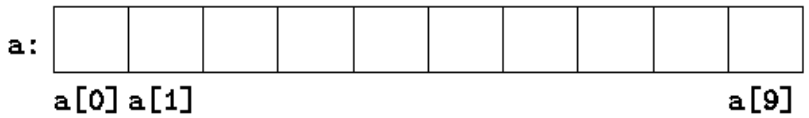
- Identifier of an array is equivalent to the address of its first element.

```
int numbers[20];
int *p;

p = numbers;      // Valid
numbers = p;      // Invalid
```

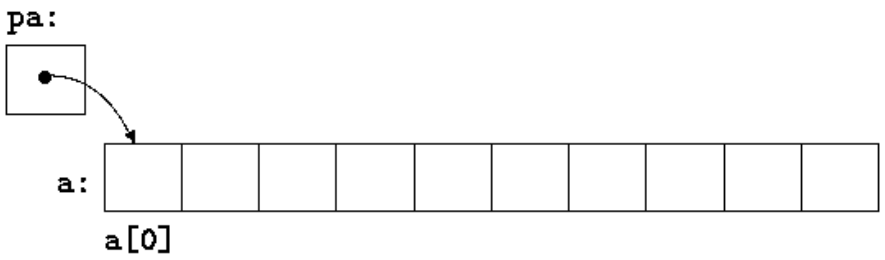
- **p** and **numbers** are equivalent and they have the same properties.
- Only difference is that we could assign another value to the pointer **p** whereas **numbers** will always point to the first of the 20 integer numbers of type int.

Pointers and Arrays: Example

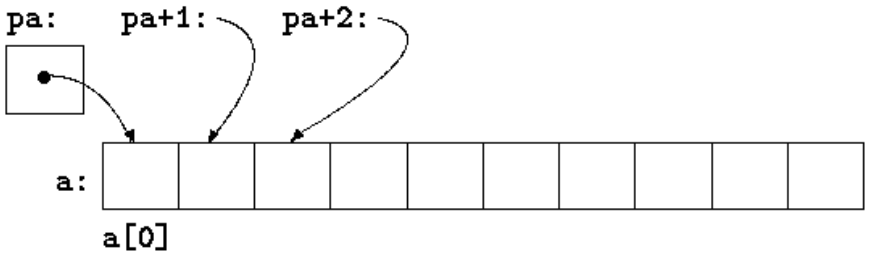
```
int a[10];
/* Init a[i] = i */
```




```
int *pa;
pa = &a[0]
x = *pa;
/*same as x = a[0]*/
```




```
int y, z;
y = *(pa + 1);
z = *(pa + 2);
```



Pointers and Arrays: More Examples

```
int a[10], *pa;
pa = a;
/* same as pa = &a[0]*/
pa++;
/*same as pa = &a[1]*/

a[i]  ⇔    *(a+i)
&a[i] ⇔    a+i
pa[i] ⇔    *(pa+i)
```

Notes

`a = pa; a++;` are **illegal**.
Think of a as a constant, not a modifiable variable.

`p[-1], p[-2],` etc. are syntactically legal.

Computing String Lengths

```
/* strlen: return length of string s */
int strlen( char *s )   /* or (char s[]) */
{
   int n;
   for ( n = 0; *s != '\0', s++ )
       n++;
   return n;
}

Callers:
strlen( "hello, world” ); /* string constant */
strlen( array ); /* char array[100]; */
strlen( ptr );  /* char *ptr; pointing to an array */
```

Passing Sub-arrays to Functions

- It is possible to pass part of an array to a function, by passing a pointer to the beginning of the sub-array.

```
my_func( int ar[ ] )  {...}
```

or

```
my_func( int *ar )  {...}
```

```
my_func(&a[5])
```

or

```
my_func(a + 5)
```

Arrays Passed to a Function

- Arrays passed to a function are passed by reference.
- The name of the array is a pointer to its first element.
- Example:

```
copy_array( int A[ ], int B[ ] );
```

- The call above does not copy the array in the function call, just a *reference* to it.

Address Arithmetic (5.4)

Given pointers p and q of the same type and integer n, the following pointer operations are legal:

- p + n, p – n
  - n is scaled according to the size of the objects p points to. If p points to an integer of 4 bytes, p + n advances by 4*n bytes.
- q – p, q – p + 10, q – p + n (assuming q > p)
  - But p + q is illegal!
- q = p; p = q + 100;
  - If p and q point to different types, must cast first. Otherwise, the assignment is illegal!
- if (p == q), if (p != q + n)
- p = NULL;
- if (p == NULL), same as if (!p)

Address Arithmetic: Example

```
/* strlen: return length of string s */
int strlen(char *s)
{
  char *p = s;
  while (*p != '\0')
     p++;
  return p - s;
}
```

Address Arithmetic: Summary

- Legal:
  - assignment of pointers of the same type
  - adding or subtracting a pointer and an integer
  - subtracting or comparing two pointers to members of the same array
  - assigning or comparing to zero (NULL)
- Illegal:
  - add two pointers
  - multiply or divide or shift or mask pointer variables
  - add float or double to pointers
  - assign a pointer of one type to a pointer of another type (except for void *) without a cast

Character Pointers and Functions (5.5)

- A *string constant* (“hello world”) is an array of characters.
- The array is terminated with the null character '\0' so that programs can find the end.

```
char *pmessage;
pmessage = "now is the time";
```

  - assigns to **`pmessage`** a pointer to the character array. This is *not a string copy; only pointers are involved.*
  - C does not provide any operators for processing an entire string of characters as a unit.

Important Difference between ...

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
```

- **amessage** will always refer to the same storage.
- **pmessage** may later be modified to point elsewhere.

Example: String Copy Function

```
/* strcpy: copy t to s; array
   subscript version */
void strcpy(char *s, char *t)
{
  int i;
  i = 0;
  while ((s[i] = t[i]) != '\0')
    i++;
}
```

```
/* strcpy: copy t to s; pointer
   version */
void strcpy(char *s, char *t)
{
  int i;
  i = 0;
  while ((*s = *t) != '\0') {
    s++; t++;
  }
}
```

```
/* strcpy: copy t to s; pointer
   version 2 */
void strcpy(char *s, char *t)
{
while ((*s++ = *t++) != '\0') ;
}
```

Dynamic Memory Allocation

CSE 2031

Fall 2012

Dynamic Memory Allocation (7.8.5)

How to allocate memory during run time?

```
int x = 10;
int my_array[ x ];   /* not allowed in C */
```

malloc()

- In stdlib.h

```
void *malloc( int n );
```

- Allocates memory at run time.
- Returns a pointer (to a void) to at least *n* bytes available.
- Returns null if the memory was not allocated.
- The allocated memory is not initialized.

calloc()

```
void *calloc( int n, int s );
```

- Allocates an array of *n* elements where each element has size *s*;
- calloc( ) initializes the allocated memory all to 0.

realloc()

- What if we want our array to grow (or shrink)?

```
void *realloc( void *ptr, int n );
```

- Resizes a previously allocated block of memory.
- **`ptr`** must have been returned from a previous **`calloc`**, **`malloc`**, or **`realloc`**.
- The new array may be moved if it cannot be extended in its current location.

free()

```
void  free( void *ptr )
```

- Releases the memory we previously allocated.
- **`ptr`** must have been returned from a previous **`calloc`**, **`malloc`**, or **`realloc`**.
- C does not do automatic “garbage collection”.

Example

```
#include<stdio.h>
#include<stdlib.h>
main() {
    int *a, i, n, sum=0;
    printf( “Input an aray size  “ );
    scanf( “%d”, &n );
    a = calloc( n, sizeof(int) );
    /* a = malloc ( n * sizeof(int) ) */
    for( i=0; i<n; i++ ) scanf( “%d”, &a[i] );
    for( i=0; i<n; i++ ) sum += a[i];
    free( a );
    printf(“Number of elelments = %d and the sum is %d\n”,n,sum);
}
```

Next time ...

- Structures (Chapter 6)