Pointer Arithmetic for Arrays

- Suppose we request enough space from malloc() or calloc() to store more than one variable of some type, and assign some value to the first element in the allocated memory:

```
int *p;
p = malloc (10 * sizeof(int));
*p = 100;     /* assigns 100 to the first sizeof (int) bytes */
```

- Since malloc() and calloc() return a pointer *to the first byte* of the allocated memory, how do we access the rest of the space beyond the first element?

Pointer Arithmetic

- We can use pointer arithmetic to access the elements beyond the first element (or even the first element).
- If p points to the first integer in our example, p + 1 points to the second integer, p + 2 points to the third, and p + n points to the $(n + 1)^{th}$.
- When the code is compiled, the compiler can generate instructions to access the appropriate bytes, because we assigned the pointer to the allocated storage to int *p, so the compiler knows the elements are integers, and it also knows sizeof (int) for the system.

Pointer Arithmetic

- **IMPORTANT**: When we do arithmetic with pointers, the value added or subtracted is *scaled* by the compiler using sizeof(type). In other words, if we add or subtract *n* from the address in a pointer variable *ptr*, the compiler will generate an instruction which adds or subtracts *n * sizeof(*ptr)*
- When you use pointer arithmetic, ***do not scale the integer*** you add or subtract by the size of the type, or you will get double scaling (this will always cause access errors).

Pointer Arithmetic cont.

- If the address of the first integer is 0x1000 (hex 1000), then the address of the second is:

  (0x1000 + 1 * sizeof(int))

- So, if sizeof(int) is 4 bytes on the system, then the address of the second integer is 0x1004.
- The address of any other element in the allocated storage can be calculated in the same way.

Memory is a sea of…

Memory

	Address	**Value in hex**	
p is pointer to int			
	1009		
	1008		Values are stored
	1007	07	in little-endian
	1006	05	byte order
	1005	03	
p+1 = 0x1004	1004	01	p[1] = 0x07050301
	1003	04	
	1002	03	
	1001	02	
p = 0x1000	1000	01	p]0] = 0x04030201

Higher addresses are at the top. Lower addresses are at the bottom.

Dereference operator with pointer arithmetic

- How can we use pointer arithmetic and the dereference operator to access elements of our dynamically allocated storage?

```
int *p;
p = malloc (10 * sizeof(int));
*p = 100;     /* assigns 100 to the first (sizeof) int bytes */
```

- To assign int values to the next two elements:

```
*(p + 1) = 200;    /*Remember, the compiler scales the value added*/
*(p + 2) = 300;
```

- More generally, for any statically or dynamically allocated array:

  array[i] accesses the same element as *(array + i)

Pointer Arithmetic cont.

- We can treat the allocated memory space as an array, because it is dynamically allocated storage consisting of a number of contiguous bytes and the elements contained in it can be accessed exactly like those in a statically allocated array.
- Thus, we can also use subscripts or indexes, as we do for statically allocated arrays. For example, we can access the third integer in the space allocated by malloc(), and pointed to by p, with p[2].
- We can access elements in statically allocated arrays in C using either of these methods as well, i.e., either with subscripts or with pointer arithmetic.
- Access via the second method may be easier for those coming from Java and C++, however, both methods will be required to do well on the midterm. ☺

Pointer Arithmetic – Caution!

- Be careful when you use the dereference operator with a pointer to elements of a statically or dynamically allocated array, along with pointer arithmetic (suppose p points to the 1st of 5 integers):

```
*(p + 3) = 45;    /* Assigns 45 to 4th int */
*p + 3 = 45;      /* Invalid – Why? */
```

Explanation of the problem

- What appears on the left side of an assignment operator in C has to be an ***L- value***, that is, **a location in memory** where a value can be stored.
- What appears on the right side of an assignment operator in C has to be an ***R-value***, that is, a numeric value which can be stored in a binary form.
- In the invalid expression on the previous slide, we have:

```
*p + 3 = 45;
```

*p + 3 is not an L-value, however, because it is not a location in memory!

Checking Bounds

- When you use pointers to access dynamically allocated storage, be sure that you do not use a pointer value that will attempt to access space outside the allocated storage!
- This will result in a segmentation fault *if you are lucky*.
- As we stated before, C has no library function which returns the size of an array, so you have to keep track of it explicitly, and pass it as a parameter to any function which accesses elements of an array (statically or dynamically allocated).

Function Parameters and Pointers

- When functions are called in C, the parameters to the function are *passed by value*:

```
int a = 5;
int b = 10;
func1(a, b);
```

- What this means is that the values of a and b in the calling environment will be passed to func1, but func1 will not have access to the memory where a and b are stored in the calling function, so it cannot alter their values.
- The ***values*** of the parameters are placed on the stack (the values are copied from the variables in the calling function, and written to the stack), before the function begins execution.

Function Parameters and Pointers cont.

- Normally, it is desirable that the called function not be able to change the values of variables in the calling environment, because this limits the interaction between the calling function and the called function, and makes debugging and maintenance easier.
- At times, though, we may want to give a function access to the memory where the parameters are located.
- In some cases in C, this is the only way we can pass a parameter; for example, elements of an array cannot be passed by value (unless each of the elements is passed as an individual parameter).
- In such cases, we can, in effect, *pass by reference*, which means we pass *a pointer to the parameter*.
- This will allow the function to alter the variable which is used to pass the parameter's value *in the calling environment*.

Example

- Consider a simple function to swap, or exchange two values, with the following declaration:

```
void swap(int x, int y);
```

Example – When pass by value does not work

- Consider using this function to swap, or exchange two values, in the following code:

```
/* Recall the declaration of the function: void swap(int x, int y); */
    int a = 10;
    int b = 5;
    swap(a, b);
```

- Even if swap correctly exchanges a and b in the called function, swap(), the value of a and b in this, the calling function will still have their original values
- What to do?

Passing a reference

```
int a = 10;
int b = 5;
swap(&a, &b);   /*Now, swap will be able to exchange
the values of a and b in the                                calling environment */
```

NOTICE: The declaration of swap must be changed too:

```
void swap(int *x, int *y);
```

because we are now passing 2 8-byte addresses rather than 2 4-byte integers *Those 8-byte references are still, as always, passed by value!*

Passing arrays as parameters

- Suppose we want to call a function declared as:

  *int sum(const int *array, int size*); It sums the elements of an array given the address of the start of the array and its size:

```
int array[6] = {18, 16, 15, 20, 19, 17};
int size = 6;
int total;
. . . .
total = sum(array, size);      /* OR total = sum(&array[0], size);  */
. . . .
```

- Any time a pointer is passed as a parameter, if the function will not write to variables pointed to by the pointer, the const keyword should be used. This is why the first parameter of sum should be declared with the const keyword above. It will allow the function sum() to read values from the array, but not affect the values in any way.