Faculty of Computer Science, Dalhousie University *29-Oct-2018*

CSCI 2132 — Software Development

Lecture 23: Pointers and Arrays (Pointer Arithmetic)

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Review of matrial from Lab 7:
  - Introduction to 'make' and Makefile
  - Review of history of Version Control Systems (rcs, cvs, Subversion, git)
  - Introduction to git, github, and GitLab
- Review of pointers
- Finished statistics.c example

Using `const` to Protect Arguments

- Passing pointers as arguments is usually done for function to make change to the caller variables
- Another reason: efficiency
- We may want to prevent accidental change to the arguments
- Example:
```
void f(const int *p) {
  /* The function is not allowed
     to modify *p */
}
```

19.5 Pointers and Arrays

In C, pointers and arrays are closely related. In fact, the array name itself is a pointer to the 0th element of the array. That is why if a function changes the element of an array parameter, the change is reflected in the corresponding argument. The function knows the memory address of the array argument and it is working with the original, not a copy of the array. This also means that we can use operators for pointers to access array elements. Let us say that we have an array of length 10, and we use a pointer p to point to its 0th element:

```
int a[10];
int *p = &a[0];
```

The second line of the above program is equivalent to:

```
int *p = a;
```

This is because `a` stores `&a[0]`. To assign the value 4 to `a[2]`, the following two statements are equivalent:

```
a[2] = 4;
```

and

```
*(a+2) = 4;
```

19.5.1 Pointer Arithmetic: Adding Pointer $+$ Integer

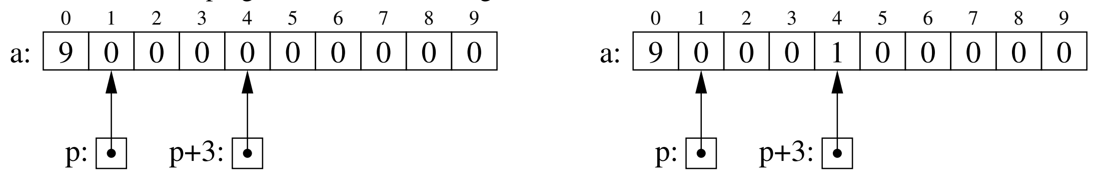
Some arithmetic operators can also be applied to pointers that point to array elements.

First, we can add an integer to a pointer. If pointer `p` points to `a[i]`, then `p+j` points to `a[i+j]`. In other words, `p+j` points to an element that is `j` positions forward in the same array. Check the following example:

```
1: int a[10] = {9};
2: int *p = &a[1];
3: (*(p+3))++;
4: printf("%d %d\n", a[1], a[4]);
```

What is the output of this program? In line 1, we initialize a so that it has values: 9, 0, 0, 0, 0, 0, 0, 0, 0, and 0. We then make `p` point to `a[1]`. Therefore, `p+3` points to `a[1+3]` which is `a[4]`. Thus line 3 increments `a[4]` by 1. Hence the output is: `0 1`

Drawing a figure can be a very helpful tool when working with pointers. The following simplified figure shows the status after the program finishes executing:



Similarly, we can also subtract an integer from a pointer. If pointer `p` points to `a[i]`, then `p-j` points to `a[i-j]`.

19.5.2 Pointer Arithmetic: Pointer $-$ Integer and Pointer $-$ Pointer

We can also subtract one pointer from another, if they point to elements in the same array. If `p` points to `a[i]` and `q` points to `a[j]`, then `p-q` is equal to `i-j`. For example, in

```
int a[10];
int *p = &a[0];
int *q = &a[5];
printf("%d\n", p-q);
```

The output of this program should be `-5`.

19.5.3 Pointer Comparison

When two pointers point to elements of the same array, we can also compare them. The result of comparison is the same as the result of comparing the indices of the elements that they point to. For the two pointers `p` and `q` defined in the previous example, we can add the following line of code:

```
printf("%d\n", p < q);
```

The result is the same as the value of `0 < 5`, which is `1`. Therefore, the output of this statement should be: `1`

What will happen if we perform pointer arithmetic on pointers that do not point to array elements? This will result in undefined behavior.

19.5.4 Examples with Pointers

We can also use p to access an element of array a. The following two statements are also equivalent to each other:

```
*(p+3) = 5;
```

and

```
p[3] = 5;
```

Thus both array subscripting and pointer arithmetic can be used on array names and pointers. To see a more complex example, we know that the following loop can be used to set all elements of a to 0:

```
int i;
for (i = 0; i < 10; i++)
   a[i] = 0;
```

This is equivalent to:

```
for (p = a; p < &a[10]; p++)
   *p = 0;
```

Note the &a[10] here. This does not point to a valid array element, but since we are using <, and not <=, it works okay. This is also equivalent to:

```
for (p = a; p < a+10; p++)
   *p = 0;
```

Now you may wonder what is the difference between an array name and a pointer. There is one difference: You cannot make an array name point to a different element of the array. It always points to the 0th element.

One issue that we need pay attention to is that the increment (++) and decrement (--) operators have higher precedence than the indirection operator (*). Thus, when we write programs that use pointers to access array elements exclusively, sometimes we need use brackets to change the precedence.

When we declare array parameters for functions, we can use pointers instead. For example, if we want to write a function that returns the largest element in an array, instead of declaring an array parameter, we can also use:

```
int max_array(int *a, int len);
```

From all the examples above, we can see that to access array elements, we can either use pointer arithmetic or array subscripting. Which approach is superior? It has been considered a few years back that the pointer arithmetic is superior. This is because the machine instructions of incrementing a pointer is more efficient than subscripting which adds an integer to the address stored in the array name. However, modern compilers are able to optimize the final code so that both approaches are about equally efficient. You can feel free to use either approach.

You are still required to learn how to use pointer arithmetic to access array elements. This is because most existing C programs use pointer arithmetic instead of array subscripting, especially those written before there was much progress in compiler optimization. You might also find that pointer arithmetic is very natural: For example, in loops when we are iterating over all elements of an array, it is sufficient to set pointer to the first element and then just use the pointer, and increment it as needed. When working with an array we generally need to have in mind two variables: the array name and the index of the array.

Most compilers do not perform compiler optimization by default, as this uses much CPU time. Normally when we write programs, we do not enable compiler optimization, and before we release software products, we do that. When using gcc the option -O can set different levels of optimization. For example, -O3 turns on the 3rd level of optimization, which is a quite high level.

19.6 Mergesort Revisited

We will now look at a Mergesort algorithm implementation using pointer arithmetic to access the elements of the given array. The code can be found at:
`~prof2132/public/mergesort2.c`

It is a fill-in-the-blanks code, and here is a copy of it:

```
/* Program: mergesort2.c */
#include <stdio.h>

void mergesort(int *lower, int *upper);
void merge(int *lower, int *middle, int *upper);

int main() {
  int len, *p;

  printf("Enter the length of the array: ");
  scanf("%d", &len);

  int array[len];

  printf("Enter %d integers:\n", len);

  for (p = array; p < array+len; p++)

    scanf("%d", _______________ );

  mergesort(&array[0], &array[len-1]);

  printf("The sorted array is: \n");
  for (p = array; p < array+len; p++) {

    printf("%d", ____________ );

    if (p < array + len - 1)
      printf(" ");
  }
  printf("\n");

  return 0;
}

void mergesort(int *lower, int *upper) {
  if (lower < upper) {
    int *middle = lower + (upper-lower)/2;
    mergesort(lower, middle);
    mergesort(middle+1, upper);
    merge(lower, middle, upper);
  }
}
```

```
/* middle belongs to lower sub-array */
void merge(int *lower, int *middle, int *upper) {
  int len_left = middle - lower + 1;
  int len_right = upper - middle;
  int left[len_left], right[len_right];
  int *p, *l, *r;

  for (l = left, p = lower; ______________ ; )

    ________________ ;

  for (r = right; _______________ ; )

    ________________ ;

  for (l=left, r=right, p=lower;
       l < left+len_left && r < right+len_right; ) {

    if ( ____________ ) ____________ ;

    else                ____________ ;

  }

  while (l < left + len_left)

    ________________ ;

  while (r < right + len_right)

    ________________ ;
}
```