Faculty of Computer Science, Dalhousie University *24-Oct-2018*

CSCI 2132 — Software Development

Lecture 21: Pointers

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

Previous Lecture

- Program organization
- External (global) variables
- Pointers: History

19 Pointers

Pointers are one of the very important features of C. Historically, their invention is credited to Harold Lawson, who introduced them in the programming language PL/I in 1964. They were also used in Pascal in 1970, but their functionality there is more restricted than in C.

The pointers allow us to refer to a variable by its memory address, and are thus crucial to memory management. To recall what a memory address means: in most computers, memory can be viewed as a sequence of bytes, and each byte of memory has a unique address which is an integer starting from 0. This is the address that we are talking about. Based on this, we can define what a pointer is.

19.1 Pointer Variables

A pointer is a variable that stores a memory address. To declare a pointer variable, we follow the following syntax:

*type *pointer_name;*

The word *type* is a variable type, and *pointer_name* is a pointer variable that can store the address of (i.e. point to) data of said type.

For example, we can define the following pointers:

```
int *p;
char *p;
int **r;
```

What is interesting above is the pointer `r`. It is a pointer to pointers to int variables. These kinds of pointers are quite frequently used in practice.

One important rule is that every pointer variable can point only to objects of a particular type. This type is called the *reference type.*

After we declare a pointer, to use it to store an address of an object, we need use the *address operator* `&`. (When we say an 'object' here, we do not mean an object in the object-oriented programming sense, but simply a construct in a memory, such as a variable.) This operator can obtain the address of a variable of any type. This address can then be assigned to a pointer that points to variables of the same type. Let us look at the following example:

```
int i, *p;
p = &i;
```

This piece of code makes p store the address of variable i. That is, p points to i. We can also initialize a pointer when we declare it. Thus the following piece of code is equivalent to the previous one:

```
int i;
int *p = &i;
```

Once we store the address of a variable in a pointer, we can use this pointer to access the value of this variable, by using the *indirection operator* '`*`'. This operator can be used to access the variable that a pointer points to. The following is a simple example:

```
int i = 7;
int *p = &i;
printf("%d\n", *p);
```

Let us look at another example:

```
1: int i = 7;
2: int *p = &i;
3: (*p)++;
4: printf("%d %d\n", i, *p);
```

In the third line of this example, we use `*p` to access the variable that `p` points to. In this case, this means we use `*p` to access `i`. Thus `(*p)++` is equivalent to `i++`. Therefore, the output of this program fragment should be:

```
8 8
```

19.2 Common Bugs with Pointers

Slide notes:

Common Bugs with Pointers

1. **Dereferencing an un-initialized pointer**
   - Result: undefined behaviour
   - Example: `int *p; *p = 5;`
2. **Dangling pointer**
   - Accessing an object that does not exist any more on stack or heap
   - Example:
     ```
     int* f() { int i=4; return &i; }
     ...
     int *p; p = f(); ++(*p);
     ```

Dereferencing un-initialized pointer: A common mistake with pointers is that we dereference pointer before it is initalized. For example, if we declare a pointer `p`, and do not assign any value to it it will likely contain some random value. If we try to use the expression `*p`, we will likely attempt to access a memory address that our program has not permission to access, which typically generates an error called Segmentation Fault. An even worse option is that we modify some memory address that we do have a right to modify, but we never intended to modify, which typically leads to difficult bugs.

Dangling pointer bug: Another kind of bugs associated with pointers is that we use a pointer whose address was valid at some point but should not be accessed any more. For example, this may happen if we return a pointer value from a function, and this pointer points to an address of an object on the stack that does not exist any more once we return from the function. When we learn about the heap memory and memory allocation, we will see that another similar situation may appear if we release memory on the heap, and later try to access it using a pointer. This problem is generally known as the *dangling pointer* problem.

19.3 Pointer Assignment

We can use the assignment operator (=) to copy pointers of the same type. An important rule to have in mind is that we cannot assign a pointer variable to another pointer of a different type. The assignment operator copies an address stored in one pointer to another pointer, and as a result both pointers will point to the same object in memory. Let us look at an example:

```
int i = 8, j = 15;
int *p = &i;
int *q;
int *r = &j;

*r = *p;
q = p;
(*q)++;

printf("%d %d %d %d %d\n", i, j, *p, *q, *r);
```

What is the output of the above program fragment? When we declare pointers, we also make `p` point to `i` and `r` point to `j`. The statement `*r = *p;` is not a pointer assignment. It assigns the value of `i` to `j`, as `p` and `r` point to `i` and `j`, respectively. The statement `q = p;` is a pointer assignment, which makes `q` to point to `i`. Thus, `(*q)++` increases the value of `i` by 1. Therefore, the output should be:

```
9 8 9 9 8
```

If you find the above analysis challenging, one strategy is to draw a figure to visualize which variable each pointer points to, as we do in class.

19.4 Pointer Arguments

Pointers make it possible for us to perform many tasks using C. Even though we have only learned a little about pointers, we can already make use of them to do a few interesting things. Let us see what we can do by using pointers as function parameters and arguments.

By using pointers as function arguments, we can modify multiple variables in the caller function. This appears to contradict the fact that in C all function arguments are passed by value. However, there is no contradiction: Even though we cannot use a function to change the value of an argument, if we pass the memory address of a variable in the caller function as a pointer argument, the statements in the function being called can reach the memory location of this variable, and make changes to what is stored in this memory location. Let us make use of this to implement a function that swaps the values of two arguments:

```
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
```

```
  *b = temp;
}
```

We would use the function as follows:

```
int a = 4;
int b = 5;
swap(&a, &b);
printf("%d %d\n", a, b);
```

This function correctly swaps the values of the variables that its pointer parameters point to. Thus, the output is `5 4`. Note the address operator (`&`) used in the function call, and the indirection operator (`*`) used in the function body.

Example: statistics.c. Let us see another example. In this example, we use a function that computes the minimum, maximum, average and standard deviation of the values stored in an array. The code can be found at:
`~prof2132/public/statistics.c`
It is fill-in-the-blanks code, and it is listed here:

```
/* Program: statistics.c */
#include <stdio.h>
#include <math.h>    /* Remember: gcc -lm statistics.c */

#define LEN 10

void statistics(double array[], int len, double* min, double* max,
double* average, double* stddev);

int main() {
  double array[LEN] = { 27.8, 31.235, 18.9,  32.55, 20.3,
                        36.0, 49.1,   29.54, 30.7,  19.5  };
  double min, max, average, stddev;

  statistics(array, LEN, ____________________________________);

  printf("minimum: %.3f\n"
         "maximum: %.3f\n"
         "average: %.3f\n"
         "standard deviation: %.3f\n", min, max, average, stddev);

  return 0;
}

void statistics(double array[], int len, double* min, double* max,
double* average, double* stddev) {
  int i;

  _________ = _________ = _________ = array[0];

  for (i = 1; i < len; i++) {

    if (array[i] < _________ )
```

```
          _________ = array[i];

      else if (array[i] > _________ )

          _________ = array[i];

      _________ += array[i];

  }

  _________ /= len;

  _________ = 0;

  for (i = 0; i < len; i++) {

      _________ += (array[i] - _________ ) * (array[i] - _________);

  }

  _________ = sqrt( _________ / len);

}
```

In the code, we used a standard C library function `sqrt` (from math.h) that computes the square root of a floating-point value. Because of this, to compile this program, we need use the `-lm` option of `gcc` (this is required by gcc, not by the C standards). So, it is important to remember to use this option whenever we use the math library. In this case the program would be compiled using the command line:

```
gcc -lm statistics.c
```

With what we have seen so far, I can now explain why in most cases, we need an address operator (`&`) in the `scanf` function call. This operator passes the address of a variable to the `scanf` function. In this way, the `scanf` function can store the data item it reads into the memory address of this variable.