Lecture 10: Pointer Applications

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/10-pointers-applications.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

In the previous lecture we have presented pointers, and in this one we will describe in what scenarios they can be useful.

Allow a Function to Access the Calling Context

Recall that arguments are passed by copy in C so with arguments of regular types a function cannot change its calling context. Pointers can be used to do so. For example here we have a function that takes an `int` pointer as parameter, then increases the pointed value by 1:

```
void add_one(int *param) {
    (*param)++;
}

int main(int argc, char **argv) {
    int x = 20;
    printf("before call, x is %d\n", x);   // print 20
    add_one(&x);
    printf("after call, x is %d\n", x);    // print 21
    return 0;
}
```

We call this function by passing as parameter the address of an `int` variable. Let's illustrate what happens. Each function stores its local variables as well as arguments somewhere in memory. We have an area for `main` and an area for `add_one`. In `main`'s area we have `x`, let's say it's located at address `0x2000`. In `add_one`'s area, when that function is called, some space is reserved for `param` and filled with `x`'s address, `0x2000`. Through `param`, `add_one` now has access to `x`'s location in memory. In effect, `add_one` can increment `x` by dereferencing `param`.

Let a Function "Return" More Than a Single Value

Pointers can be used to access the calling context when we want a function to "return" more than a single value. Here is an example in which we have a function that return both the product and the quotient of two numbers:

```
// we want this function to return 3 things: the product and quotient of n1 by n2,
// as well as an error code in case division is impossible
int multiply_and_divide(int n1, int n2, int *product, int *quotient) {
    if(n2 == 0) return -1;      // Can't divide if n2 is 0
    *product = n1 * n2;
    *quotient = n1 / n2;
```

```
    return 0;
}

int main(int argc, char **argv) {
    int p, q, a = 5, b = 10;
    if(multiply_and_divide(a, b, &p, &q) == 0) {
        printf("10*5 = %d\n", p); printf("10/5 = %d\n", q);
    }
}
```

We also want it to return an error code if the divider is 0. So we have 3 things to return. The error or success code is returned normally. And the quotient/product are returned through pointers. It works as follows: `main` reserves space for the quotient and product by creating two variables p and q, then it passes their addresses to `multiply_and_divide`, that performs the operations and write the results in `p` and `q` through the pointers.

Inefficient Function Calls with Large Data Structures Copies

Let's see another use case for pointers. Let's assume we want to write a function that updates a large struct variable that has many 8-bytes fields. Without pointers we can do it like that:

```
typedef struct {
    // lots of large (8 bytes) fields:
    double a; double b; double c; double d; double e; double f;
} large_struct;

large_struct f(large_struct s) { // very inefficient in terms of performance and memory usage!
    s.a += 42.0;
    return s;
}

int main(int argc, char **argv) {
    large_struct x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
    large_struct y = f(x);
    printf("y.a: %f\n", y.a);
}
```

The function takes the `large_struct` as parameter, updates it, and returns it. The data structure is large 6 fields times 8 bytes, so 48 bytes in total on x86-64. We know C passes parameters by copy so when `f` is called there is a first copy corresponding to `f`'s parameter `s`. Next `f` updates the struct member. An finally there is a second copy when we return the struct into `y`.

This is very inefficient both in terms of performance and memory usage. The two copy operations take time because the struct is large. And in memory there are 3 instances of the struct (`x`, `y`, and `s`) so it takes a lot of space.

Efficient Function Calls with Pointers

We can fix that issue with a pointer. The goal is to maintain a unique copy of the variable and update it in `f` though a pointer, similarly to what we saw in the previous examples. We change the parameter type to a pointer and call the function with the address of `x`. In the function we dereference the pointer before accessing the field:

```
void f(large_struct *s) {  // now takes a pointer parameter
    (*s).a += 42.0;         // dereference to access x
    return;
}

int main(int argc, char **argv) {
    large_struct x = {1, 2, 3, 4, 5, 6};
    f(&x);                  // pass x's address
    printf("x.a: %f\n", x.a);
```

```
    return 0;
}
```

When the function is called we now have the pointer which is a small parameter (64 bits i.e. 8 bytes). Only the pointer is copied when `f` is called. And in `f` we can directly update the struct through the pointer, nothing needs to be copied/returned back to `main`. It's much more efficient, we don't have to hold multiple copies of the large struct and there is no expensive copy operations.

C Arrays are Pointers

Under the hood, arrays are implemented as pointers in C. Arrays can be quite large, and it would be very inefficient to pass them by copy. Here is a function that takes an int pointer as parameter:

```
void negate_int_array(int *ptr, int size) { // function taking pointer as parameter
    for(int i=0; i<size; i++)               // also need the size to iterate properly
        ptr[i] = -ptr[i];      // use square brackets like a standard array
}

int main(int argc, char **argv) {
    int array[] = {1, 2, 3, 4, 5, 6, 7};
    negate_int_array(array, 7); // to get the pointer just use the array's name
    for(int i=0; i<7; i++)
        printf("array[%d] = %d\n", i, array[i]);
    return 0;
}
```

Notice at how we call it: we just put the name of an array variable. Also notice how we can use the square bracket on a pointer variable to address it like an array. This function negates all the elements of an int array.

Indexing Arrays with Pointers, Pointer Arithmetics

Consider the following examples:

```
int int_array[2] = {1, 2};
double double_array[2] = {4.2, 2.4};
char char_array[] = "ab";

printf("int_array[0]      = %d\n", int_array[0]);
printf("int_array[1]      = %d\n", int_array[1]);
printf("*(int_array+0)    = %d\n", *(int_array+0)); // pointer arithmetic!
printf("*(int_array+1)    = %d\n", *(int_array+1)); // +1 means + sizeof(array type)

printf("double_array[0]   = %f\n", double_array[0]);
printf("double_array[1]   = %f\n", double_array[1]);
printf("*(double_array+0) = %f\n", *(double_array+0));
printf("*(double_array+1) = %f\n", *(double_array+1));

printf("char_array[0]     = %c\n", char_array[0]);
printf("char_array[1]     = %c\n", char_array[1]);
printf("*(char_array+0)   = %c\n", *(char_array+0));
printf("*(char_array+1)   = %c\n", *(char_array+1));
```

`int_array` has 2 elements, each 4 bytes. `double_array` has 2 elements, each 8 bytes. `char_array` has 3 elements (counting the termination character), each 1 byte.

Remember that arrays elements are stored contiguously in memory. We can refer to each element of each array with either the square brackets, or the dereference operator. When we use the dereferencing operator, things work as follows. The name of the array corresponds to a pointer on the first element so dereferencing it gives the first element. The get access to the second element we increase the pointer by 1 element, hence the `+1` before dereferencing. Operations on pointers are called **pointer arithmetics** and one needs to be careful with these `int_array + 1`

is interpreted by the compiler as: `base address of int_array + the size of 1 element`. In other words, the amount of bytes that will be added by the compiler to the base address of `int_array` to determine where exactly to read in memory depend on the type of the elements contained in the array. For example for `int_array` each element is 4 bytes. For `double_array` it will be 8 bytes. And for `char_array` 1 byte.

Pointers and Data Structures

Data structures are often passed as pointers. Furthermore, structs themselves can have pointer fields. Let's take an example with a struct having two integer parameters, `member1` and `member2`, and an `int` pointer `ptr_member`:

```
typedef struct {
    int int_member1;
    int int_member2;
    int *ptr_member;
} my_struct;
```

We can create a pointer to a previously declared struct variable:

```
my_struct *p = &ms;
```

To access the int member we have two choices. The classic `*` operator for dereferencing followed by the point for field access. Don't forget the parentheses as the point takes precedence over the `*`:

```
(*p).int_member1 = 1; // don't forget the parentheses! . takes precedence over *
```

There is also a shortcut which is the arrow `->`, that can be used on a struct pointer to access a field, it first dereferences the pointer then access the field:

```
p->int_member2 = 2;    // s->x is a shortcut for (*s).x
```

One can get the address of an individual struct member with the `&` operator. Here we set the pointer field equals to the address of one of the integer fields:

```
p->ptr_member = &(p->int_member2);
```

Then we can print all the members value and dereference the member pointer:

```
printf("p->int_member1 = %d\n", p->int_member1);
printf("p->int_member2 = %d\n", p->int_member2);
printf("p->ptr_member = %p\n", p->ptr_member);
printf("*(p->ptr_member) = %d\n", *(p->ptr_member));
```

Pointer Chains

We can create pointers to pointer and construct chains. Consider this example:

```
int value = 42;          // integer
int *ptr1 = &value;      // pointer of integer
int **ptr2 = &ptr1;      // pointer of pointer of integer
int ***ptr3 = &ptr2;     // pointer of pointer of pointer of integer

printf("ptr1: %p, *ptr1: %d\n", ptr1, *ptr1);
printf("ptr2: %p, *ptr2: %p, **ptr2: %d\n", ptr2, *ptr2, **ptr2);
printf("ptr3: %p, *ptr3: %p, **ptr3: %p, ***ptr3: %d\n", ptr3, *ptr3,
        **ptr3, ***ptr3);
```

Here we have a value, and we create a first pointer to it. Then we have a second pointer pointing to the first one. The type of this pointer of pointer is `int **`, which means pointer of pointer of `int`. We also create a pointer of pointer of pointer to int, `int ***`. Note in the code the use of several star operators for dereferencing the pointers of pointers. For example `**ptr2` means get access to the value that is pointed by what is pointed by `ptr2`. In other words a first `*` gives us access to `ptr1`, and a second to `val`. Pointers of pointers are useful to create dynamically allocated arrays, as we will see in the next lecture.