Lecture 09 - Function pointers

(last updated: 2023-10-05)

Const type qualifier

We can use the const qualifier to designate a variable as read-only:

```
    const int BUFSIZE = 1024;  // good alternative to "#define BUFSIZE 1024"
```

Pointers can be const-qualified in two ways. First, we can declare a "constant pointer" like this:

```
    int * const p = &x;  // p = &y is an error, but *p = 100 is ok
```

The pointer p is stuck pointing at x. We cannot reassign p to another address. But changing the value of x, the variable that p is pointing to, is no problem.

A constant pointer stuck at a variable is not very useful, and rarely used. We will not encounter any more constant pointers in this course. A far more useful way to const-qualify a pointer is to make it a "pointer to const data":

```
    const int *p = &x;  // p = &y is ok, but *p = 100 is an error
```

In this case, we can reassign p to point to another variable all we want, but the compiler protects the data that p is pointing to. Any attempts to modify what p is pointing to will not compile.

We can, however, explicitly cast a pointer-to-const-data to a regular pointer, and modify the data that it points to through the regular pointer:

```
    int x = 5;

    const int *p = &x;

    // ++*p;  // compiler error

    int *p2 = (int *)p;  // cast away const-ness

    ++*p2;  // ok
```

Pointers to const data are often used as function parameters to document the intention that the function will not modify the pointed data. For example, the strcpy() function will declare the source pointer as a pointer to const, but not the target pointer:

```
    strcpy(char *target, const char *source);
```

Function pointers

Machine code -- the instructions that your computer hardware executes -- is stored in the code section of memory, located at the bottom of the process address space:

```
    +--------------------+
    |                    |
    |       stack        |     local variables
    |                    |
    +--------------------+
              |
              v

              ^
              |
    +--------------------+
    |                    |
    |        heap        |     malloc()ed memory blocks
    |                    |
    +--------------------+
    +--------------------+
    |                    |
    |        data        |     static variables
    |                    |
    +--------------------+
    +--------------------+
    |                    |
    |        code        |     executable machine code
    |                    |
    +--------------------+
```

Normally, knowing the addresses where different pieces of code are located in the code section is not useful -- we cannot change them because the code section is read-only. But one particular kind of addresses in that section is an exception, and they are the addresses where the functions start. They are called function pointers.

To see how function pointers are used, let us study how the qsort() library function works. The qsort() function is declared as follows:

```
    void qsort(void *base,      // the array we want to sort
               size_t nmemb,    // how many elements in that array
               size_t size,     // the size of each element in that array
               int (*compar)(const void *, const void *));  // how to compare
```

The function can sort an array of any data type, as long as it knows where the data items are located (the 'void *' parameter points to the first element of the array), how many there are, and how big each item is. The function can simply move around these data items byte by byte without having to know what they are.

But one thing that the qsort() function won't be able to do without knowing the type of the data is to compare two data items to determine whether they need to be moved or not, which is obviously essential for sorting. This is where the 4th parameter comes in. The 4th parameter is a pointer to a function that the caller supplies to qsort(). This is a function that knows how to compare two data items when called with two pointers to them. While executing its quick sort algorithm, the qsort() function repeatedly calls this function back to compare two data items. For this reason, the 4th parameter is sometimes called a "callback" function.

In the declaration of the 4th parameter:

```
    int (*compar)(const void *, const void *)
```

'compar' is the name of the parameter, and the rest of the declaration is its type. So the type of the 4th parameter is this:

```
    int (*)(const void *, const void *)
```

which is a pointer to a function that takes two arguments (whose types are both const void *) and returns an int.

The contract between the qsort() function and the caller about the callback function goes like this: qsort() will call the callback function with two pointers pointing to two data items to be compared (which will be 'void *' because qsort() does not know whether they are ints, doubles, etc.), and the callback function (that knows what those data items really are) will cast them back to typed pointers, compare the data, and return the result as negative, 0, or positive integer depending on whether the 1st item is less than, equal to, or greater than the 2nd item, respectively.

Here is an example that compares two doubles:

```
    int compare_double(const void *v1, const void *v2) {
        double x = *(double *)v1;
        double y = *(double *)v2;
        if (x < y)
            return -1;
        else if (x > y)
            return 1;
        else
            return 0;
    }
```

And another that compares two ints:

```
    int compare_int(const void *v1, const void *v2) {
        int x = *(int *)v1;
        int y = *(int *)v2;
        return x - y;
    }
```

We can pass the addresses of those functions to qsort() to let it sort different types of arrays:

```
    qsort(array_of_100_ints, 100, sizeof(int), &compare_int);

    qsort(array_of_200_doubles, 200, sizeof(double), &compare_double);
```

Note that compare_int() and compare_double() must have the exact type that qsort() asks for. Function pointers with different signatures are considered to be of distinct types. In order for us to take an address of a comparison function and pass it to qsort(), the comparison function must have the same exact type as the 4th parameter of qsort(). This is why both functions take void * parameters, and cast them to int * and double * internally.

[Exercise]

Does the following program print the 4 strings in alphabetical order?

```
        int compare_str(const void *v1, const void *v2) {
            char *s1 = (char *)v1;
            char *s2 = (char *)v2;
            // strcmp() lexicographically compares two strings
            return strcmp(s1, s2);
        }

        int main()
        {
            char *a[4] = { "hello", "abc", "xyz", "world" };

            qsort(a, sizeof(a)/sizeof(*a), sizeof(char*), &compare_str);

            for (int i = 0; i < 4; i++) {
                printf("%s\n", a[i]);
            }
        }
```

If not, why not? Can you fix it?

[Hint] Carefully draw diagrams and compare compare_str() with compare_int().

Function pointer syntax can be tricky. See the following 3 declarations:

```
    int (*f1)   (const void *v1, const void *v2);
    int  *f2    (const void *v1, const void *v2);
    int (*f3[5])(const void *v1, const void *v2);
```

Let’s consider f1 first. That’s the same one as the qsort() function’s 4th parameter, a pointer to a function that takes two pointers and returns an int.

We can assign an address of a matching function to the pointer f1 like this:

```
    f1 = &compare_double;
```

Or simply:

```
    f1 = compare_double;  // C lets us omit ’&’ for function pointers
```

We can then call compare_double() through the function pointer f1 like this:

```
    double x = 2.0;
    double y = 3.0;

    int result = (*f1)(&x, &y);
```

For function pointers, C lets us omit dereferencing, so we can write the call more simply like this:

```
    int result = f1(&x, &y);
```

The declaration for f2 simply removes the parentheses around ’*f2’. Does it make a difference? Indeed. The 2nd line is no longer a declaration for a pointer to a function, but a declaration for a function itself. That is, the 2nd line is a plain old function prototype! It says that there is a function f2 that takes two pointers and returns an ’int *’. Note that the return type is now ’int *’, which is different from the f1 case.

The 3rd line is a declaration for an array of 5 elements, where each element is the same type as f1. It is analogous to the following declaration of an array of 5 elements where each element is a ’char *’:

```
    char *a[5];
```

You can assign compare_double to an element of f3 and call it like this:

```
    f3[0] = &compare_double;  // or: f3[0] = compare_double;

    int result = (*f3[0])(&x, &y);  // or: int result = f3[0](&x, &y);
```

How are we supposed to parse a declaration as complex as the one for f3? We start from the variable name (i.e., f3), and keep adding the types according to the next operator that applies. Let’s work it out for f3 one step at a time.

```
    f3[5] - f3 is an array of 5 things

    *f3[5] - f3 is an array of 5 of something we can deference, i.e., pointers

    (*f3[5])( ... ) - once we dereference it, we call it like a function

    (*f3[5])(const void *v1, const void *v2) - f3 is an array of 5 pointers to a
                                               function that takes two pointers

    int (*f3[5])(const void *v1, const void *v2) - the result of the call is int

    So, f3 is an array of 5 pointers to a function that takes two ’const void *’
    and returns an int.
```

As we start from the variable name and keep applying the next operators, it feels like we spiral out clockwise, which is why some folks call this method of parsing C declarations a "spiral rule". But it is not a separate rule imposed by the C language. In C, postfix operators have higher precedence that prefix operators, and the "spiral rule" is simply a natural consequence of repeatedly applying the next operator. Another commonly stated (and more insightful) description of the way C declarations work is: "Declaration follows usage." That is, C declarations are written in the same way a variable will get used in an expression. That’s the way we built our description of f3’s declaration step by step.