Unit 17: Call by Value & Call by Reference

Learning Objectives

After completing this unit, students should:

- understand the differences between pure and impure functions, and understand the drawbacks of writing impure functions in terms of reasoning about the logic of code
- understand the differences between call-by-value and call-by-reference
- understand the mechanism we can perform call-by-reference in C using stack and pointers
- know the situations where call-by-reference is useful
- be able to read and write code that uses call-by-references
- know how to write Doxygen documentation to document the parameters of a function.

Pure Functions

So far, in CS1010, we have considered a very "clean" notion of functions -- each function is a black box, it takes in zero or more parameters, and it returns zero or one value. We also said that within this black box, whatever happens inside the function stays inside, and the function has no effect on the variables outside. Such "effect-free" programming leads to code that is easy to reason about and understandable. For instance, suppose we have a code like this:

```
1 long x = 1;
2 foo(x);
3 // { x == 1 }
```

We can assert that, after calling `foo`, the value of `x` is still 1.

do not incur side effects

Such functions are called *pure functions*. There is a class of programming languages, called functional programming languages, that is built entirely based on such pure functions.

As much as possible, we should write functions that are pure as it is less bug-prone, is easier to understand, and easier to prove that the code is correct.

All the functions that we have written before are invoked using a mechanism called *call by value*. This is because the *value* of the variable is *copied* onto the call stack.

Functions with Side Effects

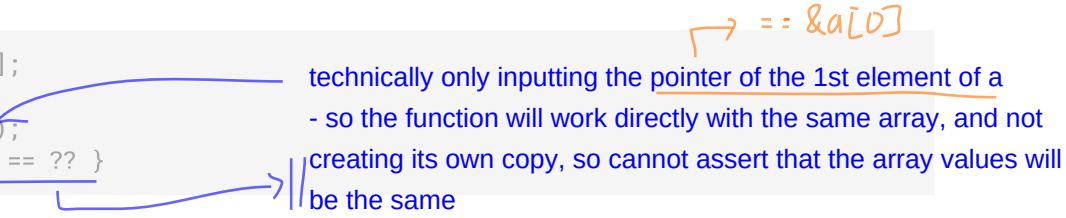
In Unit 15, we have seen a mechanism that breaks this rule. When we pass an array into a function, due to array decay, we are passing in the pointer to the first element of the array. So, what gets copied onto the call stack is the pointer, not the array. Now, with this pointer, we can modify the elements in the array directly.o

For example, the code below reset all elements in the array to 0.

```
1  void set_to_zeros(long length, long a[])
2  {
3    for (long i = 0; i < length; i += 1) {
4      a[i] = 0;
5    }
6  }
```

Whatever happens in the function no longer stays in the function. When you see code like this:

```
1  long a[10];
2  a[0] = 1;
3  foo(10, a);
4  // { a[0] == ?? }
```



We can no longer be certain that the value of `a[0]` after calling `foo` remains 1, since `foo`, or functions that it calls, could have modified `a[0]`. We have to trace through the code of `foo` and all the function it calls to understand and reason about that changes to `a[0]`.

The keyword const

Since a function taking in an array parameter can potentially modify the content of the array, as a courtesy, it is useful to add the C keyword `const` in the declaration of the array parameter:

```
1  void foo(long length, const long a[]) {
2      :
3  }
```

to signal to the readers of the code that this function only reads from the array and does not modify it. This simplifies the reasoning about the behavior of the code.

In the example below, if we know that `foo` takes in a `const` array, then we can confidently assert the value of `a[0]` after calling `foo` without reading through the code of `foo`.

```
1  long a[10];
2  a[0] = 1;
3  foo(10, a);
4  // { a[0] == 1 }
```

Call by Reference

use of pointers?

The call-by-value mechanism has its limitation. Sometimes, it is useful for a function to return more than one result. You have seen an example before in your Assignment 2, where, for the `collatz` problem, you are supposed to find both the largest stopping time and the value with the largest stopping time.

C functions, however, can only return at most one value. One way to get around this limitation is to use call-by-reference, the other is to use `struct`. We will leave the discussion of `struct` for another day, so let's see how call by reference works in this unit.

Calling by reference works by passing in the (value of a) pointer to a variable into a function, instead of the (value of) variable. Here is an example taken from the `collatz` problem.

Example: Collatz

pass in the address of the variables to compare

passing by reference

```
1  void find_max_steps(long n, long *max_n, long *max_num_steps) {
2    *max_num_steps = 0;
3    *max_n = 1;
4    for (long i = 1; i <= n; i += 1) {
5      long num_of_steps = count_num_of_steps(i);
6      if (num_of_steps >= *max_num_steps) {
7        *max_n = i;
8        *max_num_steps = num_of_steps;
9      }
10   }
11 }
```

comparing the values of the variables at that address

values that are being changed here are in main - (outside of the functions)

The method `find_max` takes in two pointers. Inside the function, we use the deference operator `*` to modify the variable pointed to by the pointers (Lines 2, 3, 7, and 8).

To use this function, we have:

```
1  int main()
2  {
3    long n = cs1010_read_long();
```

```
4     long max_num_steps = 0;
5     long max_n = 1;
6     find_max_steps(n, &max_n, &max_num_steps);
7     cs1010_println_long(max_n);
8     cs1010_println_long(max_num_steps);
9   }
```

In Line 4 above, we pass in the address of `max_n` and `max_num_steps` into `find_max_steps`. `find_max_steps` updates both variables for us.

using pointers to update the value direct

Example: Swapping Two Variables

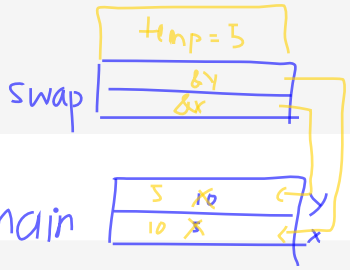
Another example of call-by-reference is a function that swaps two variables. Here is one that swaps two `long` variables.

using a function to swap variables outside of the function

pointers

```
1  void swap(long *a, long *b) {
2    long temp = *a;
3    *a = *b;
4    *b = temp;
5  }
```

dereferenced



To see `swap` in action, consider:

```
1  long a = 10;
2  long b = -4;
3  swap(&a, &b);
```

After calling `swap`, the value for `a` becomes -4, `b` becomes 10.

Documenting Call-by-Reference Parameters

uses

A parameter passed as a pointer could be used in three different ways:

- The parameter could be a read-only input, and the main purpose of passing in the value is so that the function has access to the value of the pointer.
- The parameter could be used as a vessel for the function to pass a value to the caller, similar to the parameters `max_n` and `max_num_steps` in the function `find_max_steps` above. In this case, the value contained in the variable pointed to by the pointer does not matter.
- The parameter could be used as both input and output. The value contained in the parameter is read inside the function, and the value is updated from inside the function. For example, the parameters passed to `swap` above.

documentations

In CS1010, we will be using the `Doxygen` format to document our functions. There are three types of parameters, corresponding to the three situations above: `@param[in]` is used to document a read-only parameter (note that this applies to all read-only parameters, not just pointers). `@param[out]` is used to document an output-only parameter, and `@param[in,out]` is used to document a parameter that is both input and output.

Problem Set

Problem 17.1

Complete the function `find_min_max` that takes in a `length` and an array containing `long` values of size `length`, and update the parameter `min` and `max` with the minimum and the maximum value from this array, respectively. Show how to call this function from `main`.

```
void find_min_max(long length, long array[length], long *min, long *max)
{
  :
}

int main()
{
  long list[10] = {1, 2, 3, 4, -4, 5, 6, -8, 3, 1};
    :
}
```

Problem 17.2

Consider the program below:

```
void foo(double *ptr, double trouble) {
  ptr = &trouble;
  *ptr = 10.0;
}

int main() {
  double *ptr;
  double x = -3.0;
  double y = 7.0;
  ptr = &y;

  foo(ptr, x);

  cs1010_println_double(x);
  cs1010_println_double(y);
}
```

What would be printed?