Functions With "Out" Parameters

Out Parameters

So far. `C` functions:

- take as input multiple parameters,
- use them to perform computations, and
- return **a single value** (or no value for `void` functions).

Program state. (informally) The collection of all variables declared in the program with their current values.

More formally, a **program state** is determined by the local variables present in all functions that are currently **running**.

Example. Let a `C` program have a function `int main()` that calls two functions `int first(..)` and `int second()`, then, at the beginning of execution of `int first()` the program state consists of[1]:

- All local variables declared in `int main()`;
- Values of all parameters passed to `int first()`;
- All local variables declared in `int first()`.

When `int first()` returns a value, the program state consists of only the local variables of `int main()`. After `int second()` is called, the values of the parameters passed to it, and its local variables are added to the state. When `int second()` returns a value, the state goes back to consisting only of local variables of `int main()`.

[1]Technically, an global variables declared in the program are also part of the state, but we do not use global variables in this course.

Side effect. A function produces a **side effect** during its execution if one of two things happens:

1. During the execution of the function, some output is generated and printed to an output device (e.g., terminal).
2. During the execution of the function, the value of some variable declared in another function changes: i.e., the execution of the function leads to a change in the program state that persists after the function returns a value.

Why side effects. Functions return **single, atomic values**. This limits what can be done in a function.

Example. A game piece of a computer game resides in location (x, y) on the game board. User commanded the game piece to move diagonally to location $(x + 1, y + 1)$. It is impossible to write a `C` function

```
int moveDiagonally(x,y)
```

which would update both the x and the y coordinate of the function at the same time, **without causing some side effects**.

Out Parameters

`C` allows for passage of parameters `by value` or `by reference`. I

Passing parameters by value. This is how we've been passing parameters to `C` functions so far. If a function is declared as

```
int foo(int x);
```

when a function call

```
foo(<Expression>);
```

occurs in the program, the **value** of the expression is computed and passed to the function.

Passing parameters by reference. `C` allows for another mechanism of parameter passing. If a function is declared as

```
int foo( int *x);
```

then, the following happens:

- `foo()` expects to receive the information *about the location* of some `int` variable, that is a local variable in the function that calls `foo()`.

Pointers in a nutshell. A parameter or a variable declaration of the form

```
<Type> * <Name>;
```

is known as a **pointer declaration** and the variable declared is known as a **pointer variable**.

Pointer variables **store addresses of other variables**. A variable address is a location in main memory where a variable resides.

Example. Consider the following declarations:

```
int * x;
char * y;
double * z;
```

Here, x is declared as a variable that will store an address of an integer variable; y is declared as a variable that will store an address of a char variable, and z is declared as a variable that will store an address of a double variable.

Assigning values to pointer variables. A simple assignment of the sort:

```
int * x = 4000;
```

does not work in ANSI C. Here, 4000 must be viewed NOT as a NUMBER, but rather as an address (naïvely - byte 4000 in main memory). ANSI C requires explicit conversion between addresses and integers.

Similarly, the following assignment:

```
int y = 4000;
int * x = y;
```

does not work either. Here, before the second assignment is made, the expression y is evaluated, and its value, 4000, is then attempted to be assigned to x. This, however, is the same as the attempted `int * x = 4000` assignment.

Instead, we want to assign x the value of an address of an int variable. C provides a **special operator &** for this purpose. In the code below:

```
int y = 4000;
int *x;
x = &y;
```

the second assignment should be read as follows: *set the value of **x** to be **the address** of the variable **y**.*

Passing parameters by reference. At this point, the key place where we are interested in the use of variable addresses a.k.a., variable references, a.k.a., pointers is in passing parameters to functions.

Consider the following function declaration:

```
void increase(int * x);
```

This function expects one input value, which is an address of an integer variable. To properly call this function, we must pass to it NOT the value stored in a variable, but rather the variable address. This can be done as follows:

```
int main() {
   int x;

   x = 10;
   increase(&x);

   return 0;
}
```

Working with out parameters in a function. We now need to write the code for the `void increase(int *x)` function. Suppose, we want this function to increase the value stored at the address it is passed by 10.

The following code is **INCORRECT**.

```
int increase(int *x) {
  x = x + 10;  /* x is an address, so this assignment changes the ADDRESS */
               /*    not the value stored at it                           */

}
```

`x` is an address of a variable. To change the value of `x`, is to change **the address**. Rather, we need to change the value stored at the location `x` addresses. This is done by using the **deferencing operator** `*`:

```
int increase(int *x) {
  *x = *x + 10;
}
```

Here, `*x` on the **right side of the** = is evaluated as the value stored at the address `x`. `*x` on the left side is evaluated as the variable `x` is pointing to.

scanf()

One example of the use of pointers/out parameters is the standard C library function `scanf()`.

Declared in `stdio.h`, `scanf()` reads formatted input from the *standard input stream*.

The *default standard input stream* for a `C` program is *the keyboard*. Unix's input redirection mechanisms can be used to redirect input to a file.

`scanf()` takes as input the following parameters:

- format string: the format string is similar to `printf()`. It can include text and *input value placeholders*.
- input variables: the format string specifies how many input values need to be read (count the number of `%` placeholders in the string). For each placeholder, in the order of their appearance, a `scanf()` function call must take **an address of a variable of appropriate type**.

Example. Consider the following code fragment:

```
int x;

scanf("%d", &x);
```

This is the simplest way to use the `scanf()` function. A sinlge integer value is to be read from standard input and stored in variable `int x`. To successfully perform this operation, the address of `x` needs to be passed to `scanf()`.

Format codes/placeholders. Below is the table of the `C` format codes/placeholders that can be used in `scanf()` and `printf()` statements:

Code	Type	Comment
`%d`	`int`	decimal number
`%o`	`int`	octal number (no leading '0' supplied in printf)
`%x` or `%X`	`int`	hexadecimal number
		for `printf()`, `%X` uses upper case for the digits ABCDEF)
`%ld`	`long`	decimal number (also, `%lo` and `%lx` for octal and hex)
`%u`	`unsigned`	decimal number
`%lu`	`unsigned long`	decimal number
`%c`	`char`	single character
`%s`	`char pointer`	string
`%f`	`float`	number with six digits of precision
`%g`	`float`	number with up to six digits of precision
`%e`	`float`	number with up to six digits of precision, scientific notation
`%lf`	`double`	number with six digits of precision
`%lg`	`double`	number with up to six digits of precision
`%le`	`double`	number with up to six digits of precision, scientific notation