Chapter 25: Function Parameters

Section 25.1: Parameters are passed by value

In C, all function parameters are passed by value, so modifying what is passed in callee functions won't affect caller functions' local variables.

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
    return 0;
}
```

You can use pointers to let callee functions modify caller functions' local variables. Note that this is not *pass by reference* but the pointer *values* pointing at the local variables are passed.

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

However returning the address of a local variable to the callee results in undefined behaviour. See Dereferencing a pointer to variable beyond its lifetime.

Section 25.2: Passing in Arrays to Functions

```
int getListOfFriends(size_t size, int friend_indexes[]) {
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = i;
  }
}
```

Version ≥ C99 Version < C11

```
/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
   In C11 VLAs are optional. */
void getListOfFriends(size_t size, int friend_indexes[static size]) {
```

```
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = 1;
  }
}
```

Here the `static` inside the [ ] of the function parameter, request that the argument array must have at least as many elements as are specified (i.e. `size` elements). To be able to use that feature we have to ensure that the `size` parameter comes before the array parameter in the list.

Use `getListOfFriends()` like this:

```
#define LIST_SIZE (50)

int main(void) {
  size_t size_of_list = LIST_SIZE;
  int friends_indexes[size_of_list];

  getListOfFriends(size_of_list, friend_indexes); /* friend_indexes decays to a pointer to the
                                                     address of its 1st element:
                                                                    &friend_indexes[0] */

  /* Here friend_indexes carries: {0, 1, 2, ..., 49}; */

  return 0;
}
```

See also

Passing multidimensional arrays to a function

Section 25.3: Order of function parameter execution

The order of execution of parameters is undefined in C programming. Here it may execute from left to right or from right to left. The order depends on the implementation.

```
#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d\n", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}
```

Section 25.4: Using pointer parameters to return multiple values

A common pattern in C, to easily imitate returning multiple values from a function, is to use pointers.

```
#include <stdio.h>
```

```
void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}
```

Section 25.5: Example of function returning struct containing values with error codes

Most examples of a function returning a value involve providing a pointer as one of the arguments to allow the function to modify the value pointed to, similar to the following. The actual return value of the function is usually some type such as an `int` to indicate the status of the result, whether it worked or not.

```
int func (int *pIvalue)
{
    int iRetStatus = 0;             /* Default status is no change */

    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45;   /* Modify the value pointed to */
        iRetStatus = 1;             /* indicate value was changed */
    }

    return iRetStatus;              /* Return an error code */
}
```

However you can also use a `struct` as a return value which allows you to return both an error status along with other values as well. For instance.

```
typedef struct {
    int    iStat;      /* Return status */
    int    iValue;     /* Return value */
}  RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}
```

This function could then be used like the following sample.

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Or it could be used like the following.

```
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```