Part D - Modularity

Pointers

Design procedures using selection and iteration constructs to solve a programming task
Connect procedures using pass-by-address semantics to build a complete program
Trace the execution of a complete program to validate its correctness

"Pointers are like jumps, leading wildly from one part of the data structure to another."
(Tony Hoare, 1973)

Programming languages set different rules for passing data from one module to another. The C programming language was designed from the outset to safeguard data in each module from corruption by another module. The language's pass by value mechanism prevents one function from making any direct change to any variable outside that function. A function's parameters receive *copies* of its caller's arguments so that any changes that the function makes to the parameter values only affect those copies. The calling function's arguments remain unaltered.

Cases arise that require changing the value of an external variable from within a function. The C language enables this through the variable's address.

This chapter describes how to receive the address of a variable in a function parameter, how to change the value stored in that address from within the function and how to walkthrough code that accesses addresses.

ADDRESSES

Every program variable occupies a unique address in memory throughout its lifetime. The 'address of' operator (**`&`**) applied to a variable's identifier evaluates to the address of that variable in memory.

The following program fills the address of **`x`** (**`&x`**) with user supplied input. The program then displays the value stored and its address in memory:

```
/* Working with Addresses
 * addresses.c
 */

#include <stdio.h>

int main(void)
{
        int x;

        printf("Enter x : ");
        scanf("%d", &x);
        printf("Value stored in x   :%d\n", x);
        printf("Address of x        :%x\n", &x);

        return 0;
}
```

```
Enter x : 45
Value stored in x   :45
Address of x        :12f9a0
```

`%x` is the conversion specifier for integer output in hexadecimal format.

Pointer

A variable that holds an address is called a *pointer*. To store the a variable's address, we define a pointer of the variable's type and assign the variable's address to that pointer. A pointer definition takes the form:

```
type *identifier;
```

type ******* is the type of the pointer. ***identifier*** is the name of the pointer.

The ***** operator stands for 'data at address' or simply 'data at' and is called the *dereferencing* or *indirection* operator. This operator applied to a pointer's identifier evaluates to the value in the address that that pointer holds.

The following program sotres the address of variable **x** in pointer **p** and displays the value in that address using the pointer **p**

```
/* Working with Pointers
 * pointers.c
 */

#include <stdio.h>

int main(void)
{
        int x;
        int *p = &x; // store address of x in p

        printf("Enter x : ");
        scanf("%d", &x);
        printf("Value stored in x : %d\n", *p);
        printf("Address of x      : %x\n", p);

        return 0;
}
```

```
Enter x : 45
Value stored in x : 45
Address of x      : 3cf760
```

Pointer Types

The C language supports a pointer type for every primitive or derived type:

Type	Pointer Type
`char`	`char *`
`short`	`short *`
`int`	`int *`
`long`	`long *`
`long long`	`long long *`
`float`	`float *`
`double`	`double *`
`long double`	`long double *`
`Product`	`Product *`

C compilers typically store addresses in 4 bytes of memory.

NULL Address

Each pointer type has a special value called its null value. The constant **NULL** is an implementation defined constant that contains this null value (typically, 0). This constant is defined in the **<stdio.h>** and **<stddef.h>** header files.

It is good style to initialize the value of a pointer to **NULL** before the address is known. For example,

```
        int *p = NULL;
```

PARAMETERS

A function can receive in its parameters not only data values but also addresses of program variables.

Consider a function named **internal_swap()** that swaps the values stored in two memory locations. We call this function from **main()** and note that the swap remains completely within the function itself:

```
/* Internal swap
 *  internal_swap.c
 */

#include <stdio.h>

void internal_swap (int a, int b)
{
        int c;

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

        c = a;
        a = b;
        b = c;

        printf("a is %d, b is %d\n", a, b);
}

int main(void)
{
        int a, b;

        printf("a is ");
        scanf("%d", &a);
        printf("b is ");
        scanf("%d", &b);

        internal_swap(a, b);

        printf("After internal_swap:\na is %d\n"
         "b is %d\n", a, b);

        return 0;
}
```

```
a is 5, b is 6

a is 6, b is 5

a is 5

b is 6

After internal_swap:
a is 5
b is 6
```

Although **internal_swap()** does exchange the values in **a** and **b**, the pass by value mechanism preserves the original values in **main()**.

Walkthrough Table

The walkthrough table shows how the changes remain completely within **internal_swap()**

void			int	
local_swap(int a, int b)			main(void)	
int	int	int	int	int
a	b	c	a	b
0x0012FF78	0x0012FF7C	0x0012FF6C	0x0012FF88	0x0012FF84
			5	6

5	6	?	5	6
5	6	5	5	6
6	6	5	5	6
6	5	5	5	6
6	5	5	5	6

The hexadecimal values below the variable identifiers are their addresses in memory. Note that the addresses of **a** and **b** in **internal_swap()** are different from those in **main()**.

The program copies the argument values (**a** and **b**) as initial values into parameters **a** and **b**. The swapping only affects **a** and **b** in **internal_swap()**.

Pass by Address

To change the original values, we pass the addresses of their variables instead of their values. We use these addresses to access the original values and change them from within the function.

Consider the following program

```
/* Swapping values using a function
 *  swap.c
 */

#include <stdio.h>

void swap(int *p, int *q)
{
        int c;
        c = *p;
        *p = *q;
        *q = c;
}

int main(void)
{
        int a, b;
        printf("a is ");
        scanf("%d", &a);
        printf("b is ");
        scanf("%d", &b);

        swap(&a, &b);

        printf("After swap:\na is %d\n"
               "b is %d\n", a, b);
        return 0;
}
```

```
a is 5

b is 6



After swap:
a is 6
b is 5
```

Walkthrough Table

The walkthrough table shows how the changes carry over to **main()**

void			int	
swap(int *p, int *q)			main(void)	
int *	int *	int	int	int
p	q	c	a	b
0x0012FF78	0x0012FF7C	0x0012FF6C	0x0012FF88	0x0012FF84

			5	6
0x0012FF88	0x0012FF84	?	5	6
0x0012FF88	0x0012FF84	5	5	6
0x0012FF88	0x0012FF84	5	6	6
0x0012FF88	0x0012FF84	5	6	5
0x0012FF88	0x0012FF84	5	6	5

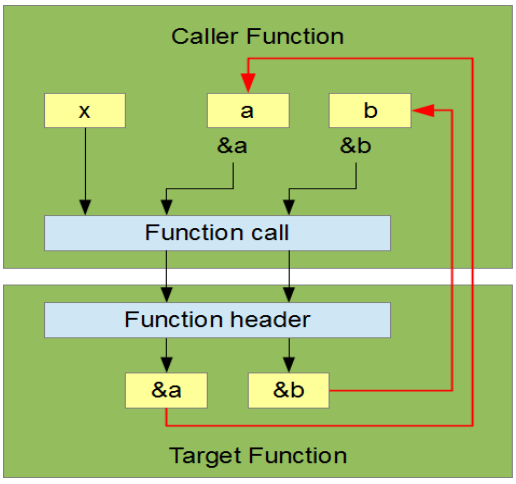
Some programmers prefer symbolic notation instead of address values. For example, they use the symbol **`main::a`** to refer to the local variable **`a`** in the function **`main()`**. A walkthrough table using symbolic notation looks something like:

`void`			`int`	
`swap(int *p, int *q)`			`main(void)`	
`int *`	`int *`	`int`	`int`	`int`
`p`	`q`	`c`	`a`	`b`
`0x0012FF78`	`0x0012FF7C`	`0x0012FF6C`	`0x0012FF88`	`0x0012FF84`

			5	6
main::a	main::b	?	5	6
main::a	main::b	5	5	6
main::a	main::b	5	6	6
main::a	main::b	5	6	5
main::a	main::b	5	6	5

MULTIPLE RETURN VALUES

C function syntax only allows for the return of a single value. If program design requires a function that returns more than one value, we do so through parameter pointers that hold the addresses of the variables that receive the multiple return values.



The following program converts day of year to month and day of month by calling function **`day_to_dm()`** to:

```
/* Day of Year to Day of Month and Month
 * day_to_dm.c
 */

#include <stdio.h>

// day_to_dm return day and month of given day in year
// assumes not leap year
```

```
//
void day_to_dm(int day, int *d, int *m)
{
        if (day < 32) {
                *m = 1;
                *d = day;
        } else if (day < 60) {
                *m = 2;
                *d = day - 31;
        } else if (day < 91) {
                *m = 3;
                *d = day - 59;
        } else if (day < 121) {
                *m = 4;
                *d = day - 90;
        } else if (day < 152) {
                *m = 5;
                *d = day - 120;
        } else if (day < 182) {
                *m = 6;
                *d = day - 151;
        } else if (day < 223) {
                *m = 7;
                *d = day - 181;
        } else if (day < 254) {
                *m = 8;
                *d = day - 222;
        } else if (day < 284) {
                *m = 9;
                *d = day - 253;
        } else if (day < 305) {
                *m = 10;
                *d = day - 283;
        } else if (day < 335) {
                *m = 11;
                *d = day - 304;
        } else if (day < 366) {
                *m = 12;
                *d = day - 334;
        }
}

int main(void)
{
        int day, d, m;

        printf("Day of Year : ");
        scanf("%d", &day);
        day_to_dm(day, &d, &m);
        printf("Day/Month is %d/%d\n", d, m);

        return 0;
}
```

```
Day of Year : 357

Day/Month is 23/12
```

Functions that return values through their parameters can reserve their return values for reporting any error codes produced by the function.

EXERCISES

- Complete the Workshop on Functions with Pointers
- Complete the exercise Pointer Walkthrough

- Complete the following practice problem:
  - Reinforcing Bar