Problem Solving and Program Design - Chapter 6

Cory L. Strope

Chapter 6

6.1 Functions with Simple Output Parameters

6.2 Multiple Calls to a Function with I/O Parameters

6.3 Scope of Names

6.4 Formal Output Parameters as Actual Arguments

6.6 Debugging and Testing a Program System

6.7 Common Programming Errors

- In `this_chapter -= 3`, we wrote the separate components – functions – of a program, corresponding to individual steps in a problem solution.
    - Providing input to a function
    - Returning a **single** output.
- In this chapter, we learn how to connect functions to create a program system – an arrangement of parts that makes your program pass information from one function to another.

Functions with Simple Outputs

`func(argument_list)`

- Argument lists provide the communication links between the `main` function and its function subprograms.
- Arguments enable a function to manipulate different data each time it is called.
- So far, we have passed inputs into a function and returned only one result value from a function.
- This section, we use output parameters to return multiple results from a function.

Functions – Simple Outputs

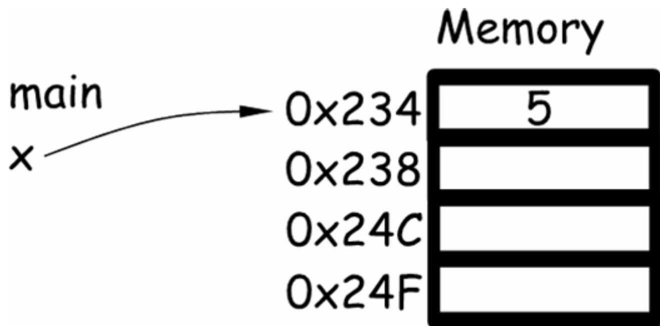
How a function call works:

- When a function call executes, the computer allocates memory space in the function data for each formal parameter.
- The value of each actual parameter is stored in the memory cell allocated to its corresponding formal parameter.
- The function body can manipulate this value.

Example Program

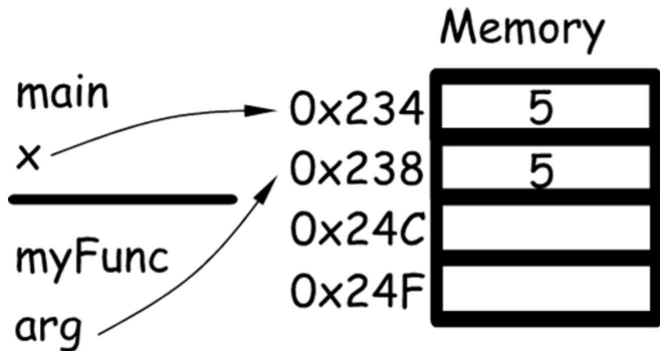
```
void myFunc(int arg);

int main(void) {
    int x = 5;
    myFunc(x);
    printf("%d\n", x);
    return 0;
}

void myFunc(int arg) {
    arg = 4;
}
```

Functions – Simple Outputs

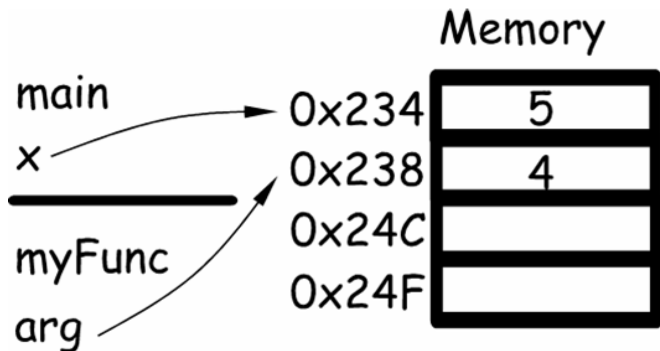



```
int main(void) {
    int x = 5;
    ...
```

Functions – Simple Outputs

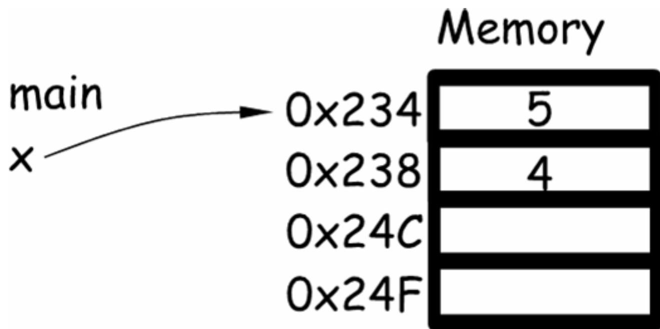



```
int x = 5;
myFunc(x);
...
```




```
void myFunc(int arg) {
    arg = 4;
}
```

Functions – Simple Outputs




```
    int x = 5;
    myFunc(x);
    printf("%d\n", x);
}
```

Output = 5

Functions with Multiple Output Parameters

- Using a $*$ in the argument list allow us to define a **pointer** to memory.
  - A pointer must have a data type for the compiler to reserve the correct amount of memory.
- To pass a memory location in the calling of the function, use a &.
  - The & is used to send the memory address of the variable.
- The $*$ is used to access the data and not the address value.

pointer.c

Example Program

```
void myFunc(int *arg);

int main(void) {
    int x = 5;
    myFunc(&x);
    printf("%d\n", x);
    return 0;
}

void myFunc(int *arg) {
    arg = 4;
}
```

Is this right?

Example Program

```
void myFunc(int *arg);

int main(void) {
    int x = 5;
    myFunc(&x);
    printf("%d\n", x);
    return 0;
}

void myFunc(int *arg) {
    arg = 4;    <---------
}
```

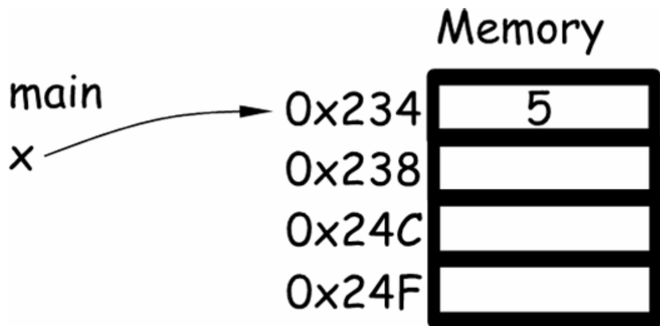
This sets the *pointer value* of `arg` to memory location 4. This will lead to the beloved error:

`Segmentation fault`

Example Program

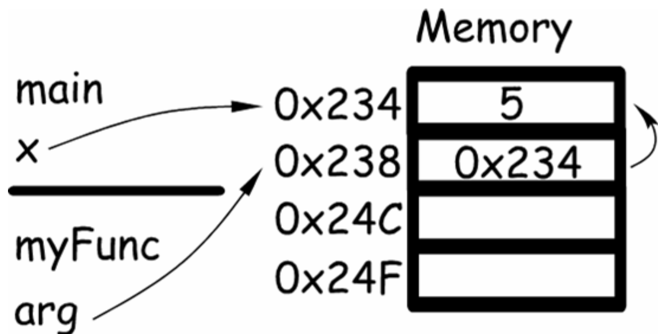
```
void myFunc(int *arg);

int main(void) {
    int x = 5;
    myFunc(&x);
    printf("%d\n", x);
    return 0;
}

void myFunc(int *arg) {
    *arg = 4;
}
```

Functions – Simple Outputs



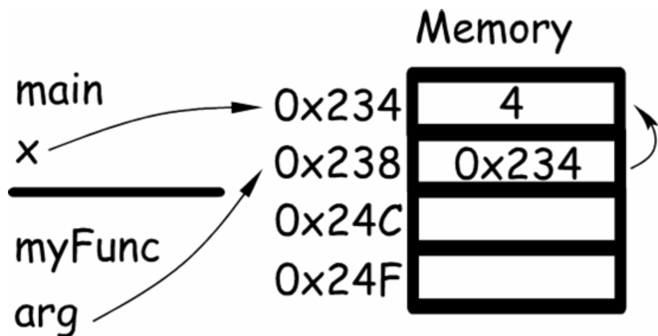

```
int main(void) {
    int x = 5;
    ...
```

Functions – Simple Outputs




```
int x = 5;
myFunc(&x);
...
```

Functions – Simple Outputs




```
void myFunc(int *arg) {
    *arg = 4;
}
```

Functions – Simple Outputs

```
    int x = 5;
    myFunc(&x);
    printf("%d\n", x);
}
```

Output = 4

- In the function declaration, variables named with $*$ are identified as *input/output parameters*.
  - The function uses the input values and may change the actual values passed.
- In this example, we pass information into a function and return the results through the functions input/output parameters.
- It also demonstrates how a function may be called more than once in a given program and process different data in each call.

```c
#include <stdio.h>
void order(double *smp, double *lgp);

int main(void) {
   double num1, num2, num3;

   printf("Enter three numbers seperated by blanks> ");
   scanf("%lf %lf %lf", &num1, &num2, &num3);

   order(&num1, &num2);
   order(&num1, &num3);
   order(&num2, &num3);

   printf("The numbers in ascending order are: ");
   printf("%.2f %.2f %.2f\n", num1, num2, num3);
   return 0;
}
```

```
void order( double *smp, double *lgp) {
   double tmp;
   if (*smp > *lgp) {
      tmp = *smp;
      *smp = *lgp;
      *lgp = tmp;
   }
}
```

Generally, functions that take a set of parameters and return a single result are preferable because of readability, such as `sin(x)` and other math functions.

Scope of Names

- The **scope of a name** refers to the region of a program where a particular meaning of a name is visible or can be referenced.
  - all people are people
  - chips $\in$ England, french fries $\in$ America, freedom fries $\in$ Washington D.C.
  - The word "fridge" for refridgerator.

Scope – Not just a Mouthwash

```
#define MAX 950
#define LIMIT 344343

void one(int anarg, double second);
int fun_two(int one, char anarg);

int main(void) {
    int localvar; ...
}

void one(int anarg, double second) {
   int onelocal; ...
   { int twolocal;  }
}

int fun_two(int one, char anarg) {
    int localvar; ...
}
```

Scope

- The name `MAX` and `LIMIT` are defined constant macros
  - Their scope begins at their definition and continues to the end of the source file.
  - All three functions can access `MAX` and `LIMIT`.
- All of the formal parameters (function arguments) and local variables are visible only from their declaration to the closing brace of the function in which they are declared, even if the names are the same.

Formal Output Parameters as Actual Arguments

- So far all of our actual arguments in calls to functions have been either local variables or input parameters of the calling function.
- However, sometimes a function needs to pass its own output parameters as an argument when it calls another function.

```
void scan_fraction(int *nump, int *denomp) {
   char slash, discard;
   int status, error;

   do {
      error = 0;
      printf("Enter a common fraction as two integers ");
      printf("separated by a slash> ");
      status = scanf("%d %c%d", nump, &slash, denomp);

      //test for errors as shown in the book

      do {
         scanf("%c", &discard);
      } while (discard != '\n');
   } while (error);
}
```

Formal Output Parameters as Actual Arguments

- So far all of our actual arguments in calls to functions have been either local variables or input parameters of the calling function.
- However, sometimes a function needs to pass its own output parameters as an argument when it calls another function.
  - `scan_fraction(int *nump, int *denomp)`
  - `scanf("%d %c%d",nump,&slash,denomp);`
- Key point is that in the scanf, there is no & in front of `nump` or `denomp`.

Debugging and Testing a Program System

- As the number of statements in a program grows, the possibility of error also increases.
  - Reducing the number of operations each function performs also reduces the likelihood of errors.
  - Small functions are also much easier to read and test.
- **top-down testing** is the process of testing a program starting at `main` function and testing each function thereafter.
- We test a function with a **unit test** by writing a short driver function to call it.
  - This can be done in the main function by commenting out other function calls (comment out the head and legs of stickman to see if the body is correctly made).
- **Bottom-up testing** is the process of separately testing individual functions before inserting them in a program system.
- **System integration tests** are tests of the entire system
- Try both top-down and bottom-up to make sure the program is fully tested.

Common Programming Errors

- Many opportunities for errors arise when you use functions with parameter lists
  - Ensuring that the actual argument list has the same number of items as the formal parameter list.
  - Each input argument must be of a type that can be assigned to its corresponding formal parameter.
  - An actual output argument must be of the same pointer data type as the corresponding formal parameter.
- Proper use of parameters is difficult for beginning programmers to master, but it is an essential skill.
- It is easy to introduce errors in a function that produces multiple results.
  - The output parameter must be of a pointer type ($*$)
  - The calling function neglects to send a correct variable address (using &)

Common Programming Errors

- An identifier referenced outside of its scope will return an undeclared symbol syntax error.
- Commonly occurs if the { or } are misplaced, or if too many open or close brackets are present in the program.