C Programming Notes

作家: Anrich Tait

Contents

Chapter 1

Examples:

Listing 1.1: Template Code Block

```c
1  #include <stdio.h>
2
3  int main(void)
4  {
5          printf("This is a template");
6          return (0);
7  }
```

This is a very basic example of how code blocks will look in the document.

Chapter 2

Functions

2.1 Outline:

The following questions/topics will be addressed:

1. What are functions?

2. How are functions utilised?

3. Examples of using functions.

2.2 Definition

Functions break large computing tasks into smaller ones. This helps make code easier to change in the future and can also help with readability. Appropriate functions also hide details of operations from parts of the program that don't need them. C programs generally consist of many small functions rather than fewer large ones. The most basic example of a function is main(), every program has one and it is executed before any other functions in the program. Functions can be used for any task but generally it is best for each function to accomplish a specific task. A function declaration provides the actual body of the function.

The standard library provides numerous builtin functions that your program can call for example:

1. strcat() to concatenate strings

2. memcpy() to copy the memory location of one parameter to another.

A FUNCtional example:

Listing 2.1: Example of calling a function in C

```c
#include <stdio.h>

int roadTravelled()
{
        printf("You became a victim of a "
                        "roadside robbery and murder.\n");
        printf("GAME OVER");
        return (1);
}
int roadLessTravelled()
{
        printf("You got lost in the woods.\n "
                        "And died of starvation \n");
        printf("GAME OVER");
        return (2);
}

int main()
{
        int answer;

        printf("You are presented with two paths, "
                        "the road often travelled by "
                        "others, and the road less "
                        "travelled. Neither offers any "
                        "indication of what lays ahead, which "
                        "will you choose?\n");

        printf("1 = Road often travelled\n "
                        "2 = Road less travelled\n");
        scanf("%d", &answer);

        if (answer == 1)
        {
                roadTravelled();
```

```
36          }
37          else if (answer == 2)
38          {
39                  roadLessTravelled ();
40          }
41 }
```

The above example gives the user two choices and then asks for input. based on
the input a specified function is called to output the relevant text. While the use of
functions is explicitly necessary in this example it shows how code can be broken up
into smaller blocks, each with a specific purpose.

2.3 Defining a function

Syntax:

```
return_type function_name(parameter list)
{
        body of function
}
```

Here is a break down of each part of this syntax:

1. return_type: The type of value the function will return.

2. function_name: Name of the function.

3. (parameter list): list of parameters passed to the function.

4. body of function: code to be executed

Example:

```
int add(int a, int b)
{
        int result = a + b;
        return (result);
}
```

Break down:

1. int: Return type of function (integer).

2. add: Name of function.

3. (int a, int b) parameters.

4. int result ā + b: declare result variable and code that will be executed.

5. return (result) returns the value of the result variable.

Note: if a function does not return anything it can be specified as void.

Functions can be called from other parts of a program by using the function name and passing in the required parameters.

```
int x = 5;
int y = 7;

int z = add(x, y);
```

The function 'add' is called with x and y as the parameters. The x parameter will replace 'a' and 'y' will replace 'b'. The result is stored in 'z'.

2.4 Function types

1. Functions that return a value: Perform a specific task and return a value of a specific data type using the return statement. The previous "add" function is an example of this.

```
char get_first_char(char *str)
{
        return [0];
}
```

2. Functions that return nothing: Perform a task but does not return any value.

```
void print_hello()
{
        printf("Hello, world!")
}
```

3. Functions that take no parameters: Does not take any paramters but perfomrs a specific task.

```
void clear_screen()
{
        system("clear");
}
```

4. Functions that take parameters but return nothing.

```
void greet(char *name)
{
        printf("Hello, %s", name);
}
```

2.5   Calling a function

When a function is called, control of the program is transferred to the function until it's return statement or closing brace is reached. Then the main() function is used again.

To call a function you need to pass the required parameters.

Passing parameters to a function basically means providing values or variables to the function.
Going back to the 'add' function as an example.
x and y are the parameters that are passed to the function. When the function is called with arguments the values of the arguments are assigned to the corresponding parameters. So if we call the 'add' function with arguments 3 and 4 the values 3 and 4 are assigned to x and y respectively.

2.6   Function arguments

For a function to use arguments it must declare variables that accept the values of the arguments. These variables are called the formal parameters of a function.

Formal parameters behave like other local variables and are created upon entry to the function and are destroyed upon exit.

There are two ways arguments can be passed to a function:

2.6.1   Call/Pass by value:

The function receives a copy of the argument passed, not the original argument. So any changes made to the argument inside the function do not change the original value of the argument.

Listing 2.2: Call by value example

```c
#include <stdio.h>

void increment(int x)
{
        x++;
}

int main(void)
{
        int num = 10;

        increment(num);
        printf("%d", num);
        return (0);
}
```

In this example the 'increment' function takes an integer argument ('x') and then increments it by one. The following main function calls the increment function and passes "num" to it. This does not change the output as the increment function does not change the original value "num".

2.6.2   Call/Pass by reference:

Calling a function by reference involves passing arguments to a function using the parameters address. This way the actual value is changed, not just a copy of it.

To pass a variable by reference in C, you must declare the function parameter as a pointer type using the * operator.

The following example function takes a pointer to an integer as a parameter.

Listing 2.3: Call by reference example

```c
#include <stdio.h>

void increment(int *x)
{
        /*takes a pointer to integer and increments by one*/
        (*x)++;
}

int main(void)
{
        int num = 5;

        increment(&num);

        printf("%d", num);
        return (0);
}
```

Note: by default C passes arguments by value.

2.7   Function Prototypes

2.7.1   What is a function prototype

A function prototype is a declaration of a function that describes the function's interface to the rest of the program. It specifies the function's name, return type and parameter types.

These are usually specified at the beginning of a file before any other functions.

Syntax:

```
return_type function_name(parameter_list);
```

For example:

```
int max(int numOne, int numTwo);
```

This prototype declares a funtion named max that takes two integer parameters.

2.7.2   Why are function prototypes used?

1. Type checking: By declaring the function prototype before calling the function the compiler can check that the arguments passed to the function match the expected types.

2. Avoid implicit declaration: If a function is not given an implicit type the compiler assumes it is an integer, this can lead to misleading errors relating to different return types or number of arguments.

3. Optimization: Allows the compiler to generate more efficient code by providing information about the function's interface. This can lead to smaller, faster code.

4. Documentation: Can server as documentation for the function, making it easier for other programmers to understand how to use a function.

Chapter 3

Pointers

Syntax:

<type> *<name>

1. type = is the data type that the pointer will point to.

2. name = is the name of the pointer variable.

Example:

To declare a pointer variable that will point to an integer value the following syntax is used,

```
int *ptr;
```

This declares a pointer variable named "ptr" that can point to an integer value.

To assign a value to a pointer variable the address-of operator (&) is used to get the memory address of the variable.

Example:

Assign the address of an intger variable named "x" to the pointer variable "ptr",

```
int x = 10;
int *ptr = &x;
```

This assigns the address of "x" to "ptr", so now "ptr" points to x. To access the value that is stored at the memory location pointed to by a pointer variable the dereference operator is used (*).

3.1   Concept

Pointers are a fundamental concept in the C programming language, and understanding how they work is essential for writing efficient and effective C code. A pointer is a variable that stores the memory address of another variable. By using pointers, C programs can directly access and manipulate the values of other variables in memory, providing a powerful tool for building complex data structures and optimizing program performance.

At its core, a pointer is simply a memory address. When you declare a pointer variable in C, you are creating a variable that can store the address of another variable in memory. To declare a pointer variable, you use the * symbol before the variable name, like this: int *ptr;. This declares a pointer variable named ptr that can store the memory address of an integer variable.

Once you have declared a pointer variable, you can use the & operator to get the memory address of another variable, and assign that address to the pointer variable using the assignment operator =. For example, if you have an integer variable named x, you can get its memory address with &x, and assign it to the pointer variable ptr like this: ptr = &x;. Now, ptr points to the memory location of x, and you can use the pointer to access and manipulate the value of x directly.

One of the most common uses of pointers in C is to pass variables by reference to functions. In C, when you pass a variable to a function as an argument, the function gets a copy of the variable's value. However, by passing the memory address of the variable instead, you can allow the function to directly access and modify the variable's value. This can be useful for functions that need to modify variables outside of their own scope.

Another important use of pointers in C is for dynamic memory allocation. In C, you can use the malloc() function to allocate a block of memory of a specified size at runtime. The malloc() function returns a pointer to the beginning of the allocated memory block, which you can then use to access and manipulate the memory. Once

you are done with the memory, you should free it using the free() function to avoid memory leaks.

Pointers can also be used to create complex data structures in C, such as linked lists, trees, and graphs. By using pointers to connect nodes together in these structures, you can create efficient and flexible data structures that can be easily traversed and manipulated.

However, while pointers are a powerful tool in C programming, they can also be dangerous if used improperly. Pointer errors, such as dereferencing a null pointer or accessing memory that has already been freed, can cause program crashes or even security vulnerabilities. To avoid these errors, it is important to carefully manage and validate pointers in your code, and to use best practices for pointer manipulation, such as initializing pointers to null and checking for null before dereferencing them.

In conclusion, pointers are a fundamental concept in C programming that allow you to directly access and manipulate the values of variables in memory. Pointers are used extensively in C for passing variables by reference, dynamic memory allocation, and creating complex data structures. While pointers can be a powerful tool, they can also be dangerous if used improperly, and it is important to carefully manage and validate pointers in your code to avoid errors and vulnerabilities.

Summary:

The syntax for a pointer consists of the data type followed by an asterisk and the name of the pointer variable. The address-of operator (&) is used to assign a value to the pointer, and the dereference operator (*) is used to access the value stored at the memory location pointed to by the pointer.

3.2 Examples

3.2.1 How to use pointers:

Listing 3.1: Pass variables by reference to functions:

```c
#include <stdio.h>

void swap(int *x, int *y)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}

int main()
{
        int a = 5, b = 10;
        swap(&a, &b);
        printf("a=%d, b=%d\n", a, b); //Output: a=10, b = 5
        return (0);
}
```

In this example, swap() is a function that takes two integer pointers as arguments and swaps their values. By passing the memory addresses of a and b to swap() using the & operator, we allow the function to directly modify the values of a and b.

Listing 3.2: Allocate memory dynamically: language

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

int main()
{
        int *arr = malloc (sizeof(int) * 10);
        if (arr != NULL)
        {
                for (int i = 0; i < 10; i++)
                {
                        arr[i] = 1;
                }
        }
        free(arr);
        return (0);
}
```

In this example, we use the malloc() function to dynamically allocate an array of 10 integers, and store its memory address in the pointer variable arr. We then use a loop to assign the values 0 to 9 to the array elements. Finally, we use the free() function to release the allocated memory.