Unit #: 2

Unit Name: Counting, Sorting and Searching

Topic: Functions in C

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C contructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Introduction

In case we want to repeat the set of tasks or instructions, we may have two options:

- Use the same set of statements every time we want to perform the task
- Create a function to perform that task, and just call it every time when we need to perform that task, is usually a good practice and a good programmer always uses functions while writing code in C

Functions **break large computing tasks into smaller ones** and enable people to build on what others have done instead of starting from scratch. In programming, **Function is a subprogram to carry out a specific task**. A function is a self-contained block of code that performs a particular task. Once the function is designed, it can be **treated as a black box**. The inner details of operation are invisible to rest of the program. Functions are useful because of following reasons:

○ To **improve the readability** of code.

○ **Improves the reusability** of the code, same function can be used in any program rather than writing the same code from scratch.

○ **Debugging of the code would be easier** if we use functions, as errors are easily traced.

○ **Reduces the size of the code**, redundant set of statements are replaced by function calls.

Types of functions

C functions can be broadly classified into two categories:

Library Functions

Functions which are defined by C library. Examples include printf(), scanf(), strcat() etc. We just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

User-defined Functions

Functions which are defined by the developer at the time of writing program. Developer can make changes in the implementation as and when he/she wants. It reduces the complexity of a big program and optimizes the code, supporting Modular Programming Paradigm

In order to make use of user defined functions, we need to understand three key terms associated with functions: **Function Definition, Function call and Function Declaration.**

Function Definition

Each function definition has the form

```
return_type function_name(parameters)   // parameters optional
{
        // declarations and statements
}
```

A function definition should have a return type. The return type must match with what that function returns at the end of the function execution. If the function is not designed to return anything, then the type must be mentioned as void. Like variables, function name is also an identifier and hence must follow the rules of an identifier. Parameters list is optional. If more than one parameter, it must be comma separated. Each parameter is declared with its type and parameters receive the data sent during the function call. If function has parameters or not,   but the parentheses is must. Block of statements inside the function or body of the function must be inside { and }. There is no indentation requirement as far as the syntax of C is considered. For readability purpose, it is a good practice to indent the body of the function.

Function definitions can occur in any order in the source file and the source program can be split into multiple files. One must also notice that only one value can be returned from the function, when called by name. There can also be side effects while using functions in c

```
int sum(int a,  int b)
{
        return a+b;
}

int decrement(int y)
{
        return y-1;
}
```

```
void disp_hello()       // This function doesn't return anything
{
        printf("Hello Friends\n");
}

double use_pow(int x)
{
        return pow(x,3);        // using pow() function is not good practice.
}

int fun1(int a, int)        // Error in function definition.
{  }

int fun2(int a, b)         // Invalid Again.
{   }
```

Function Call

Function-name(list of arguments); // semicolon compulsory and Arguments depends on the number of parameters in the function definition

A function can be called by using **function name followed by list of arguments** (if any) enclosed in parentheses. The function which calls other function is known to be Caller and the function which is getting called is known to be the Callee. **The arguments must match the parameters in the function definition in it's type, order and number**. **Multiple arguments must be separated by comma. Arguments can be any expression in C. Need not be always just variables. A function call is an expression. When a function is called, Activation record is created**. **Activation record is another name for Stack Frame**. It is composed of:

- Local variables of the callee
- Return address to the caller
- Location to store return
- value Parameters of the
- callee Temporary variables

The order in which the arguments are evaluated in a function call is not defined and is determined by the calling convention(out of the scope of this notes) used by the compiler. It is left to the compiler Writer. **Always arguments are copied to the corresponding parameters**. Then the control is transferred to the called function. Body of the function gets executed. When all the statements are executed, callee returns to the caller. OR when there is return statement, the expression of return is evaluated and then callee returns to the caller. If the return type is void, function must not have return statement inside the function.

Coding Example_1:

```
#include<stdio.h>
int main()
{
        int x = 100;
        int y = 10;
        int answer = sum(x,y);
        printf("sum is %d\n",answer);
        answer = decrement(x);
        printf("decremented value is %d\n",answer);
        disp_hello();
        double ans = use_pow(x);
        printf("ans is %lf\n",ans);
        answer = sum(x+6,y);
        printf("answer is %d\n", answer);
        printf("power : %lf\n", use_power(5));
        return 0;
}
```

Function Declaration/ Prototype

All functions must be declared before they are invoked. The function declaration is as follows.

return_type Function_name (parameters list); // semicolon compulsory

A function may or may not accept any argument. A function may or may not return any value directly when called by name. There are different type of functions based on the arguments and return type.

- Function without arguments and without return value
- Function without arguments and with return value
- Function with arguments and without return value
- Function with arguments and with return value

The parameters list must be separated by commas. The parameter names do not need to be the same in declaration and the function definition. The types must match the type of parameters in the function definition in number and order. Use of identifiers in the declaration is optional. When the declared types do not match with the types in the function definition, compiler will produce an error.

Parameter Passing in C

Parameter passing is always by Value in C. Argument is copied to the corresponding parameter. The parameter is not copied back to the argument. It is possible to copy back the parameter to argument only if the argument is l- value. Argument is not affected if we change the parameter inside a function.

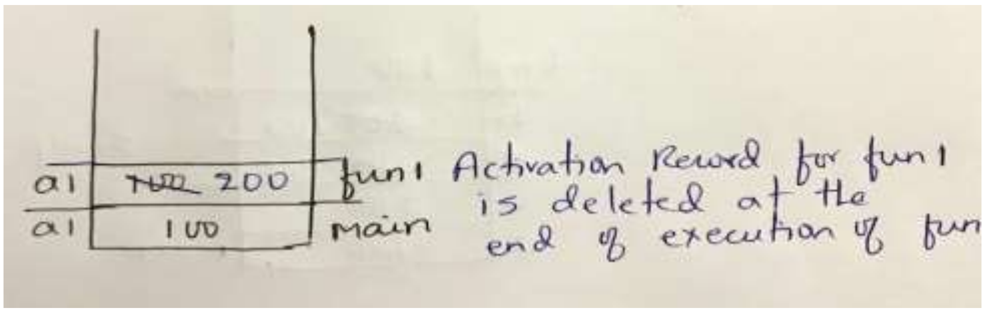
Types of parameters: Actual Parameter/Arguments and Formal Parameters

Actual parameters are values or variables containing valid values that are passed to a function when it is invoked or called. **Formal parameters** are the variables defined by the function that receives values when the function is called.

Coding Example_2:

```
void fun1(int a1); // declaration
void main()
{
        int a1 = 100;
        printf("before function call a1 is %d\n", a1); // a1 is 100
        fun1(a1);        // call
        printf("after function call a1 is %d\n", a1);    // a1 is 100
        return 0;
}

void fun1(int a1)
{
        printf("a1 in fun1 before changing %d\n", a1);          //100 a1 = 200;
        printf("a1 in fun1 after changing %d\n", a1); //200
}
```

a1 has not changed in main function. The parameter a1 in fun1 is a copy of a1 from main function. Refer to the below diagram to understand activation record creation and deletion to know this program output in detail.

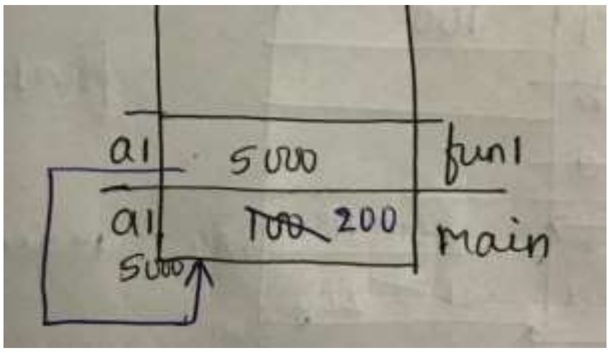

a1
~~100~~ 200
fun1
a1
100
main
Activation Record for fun1 is deleted at the end of execution of fun

Think about this Question: Is it impossible to change the argument by calling a function? Yes – by passing the l-value

Coding Example_3:

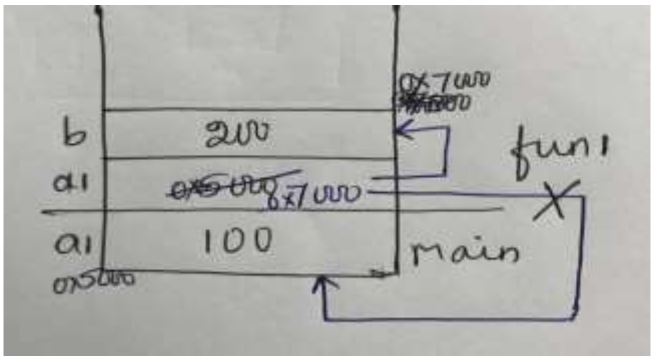
```
void fun1(int *a1);
int main()
{
        int a1 = 100;
        printf("before function call a1 is %d\n", a1); // 100
        fun1(&a1);      // call
        printf("after function call a1 is %d\n", a1);    // 200
}
```

```
void fun1(int *a1)
{
        printf("*a1 in fun1 before changing %d\n", *a1);     //100
        *a1 = 200;
        printf("*a1 in fun1 after changing %d\n", *a1);      //200
}
```



Coding Example_3:

```
void fun1(int *a1);
int main()
{
        int a1 = 100;
        printf("before function call a1 is %d\n", a1); // 100
        fun1(&a1);      // call
        printf("after function call a1 is %d\n", a1);    // 100
}
void fun1(int *a1)
{
        int b = 200;
        printf("*a1 in fun1 before changing %d\n", *a1);     //100
        a1 = &b;
        printf("*a1 in fun1 after changing %d\n", *a1);      //200
}
```

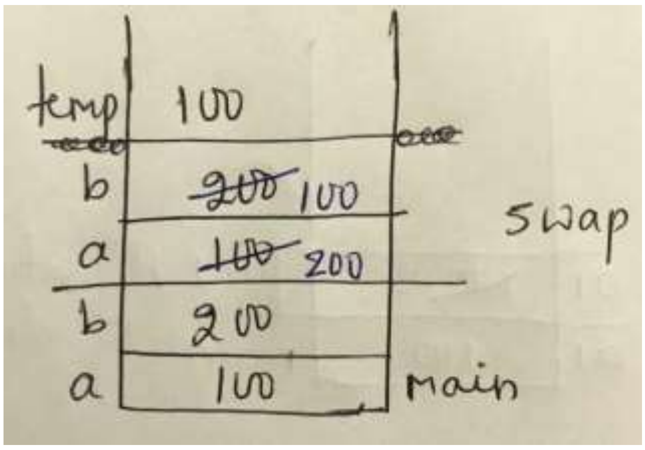


Shall we write the function to swap two numbers and test this function?

Coding Example_4:

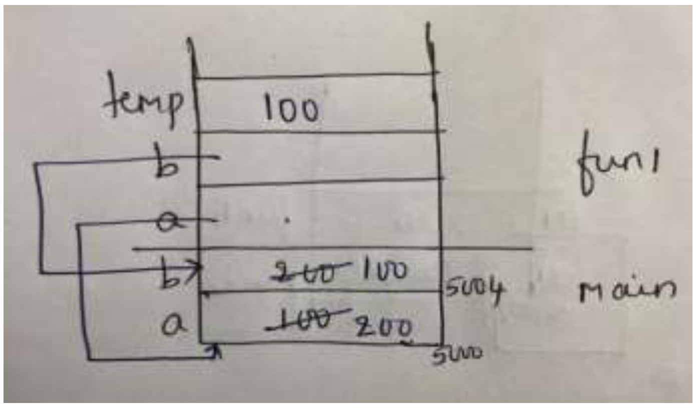
Version – 1: Is this right version?

```
void swap(int a, int b)
{
        int temp = a; a = b; b = temp;
}
int main()
{
        int a = 100; int b = 200;
        printf("before call a is %d and b is %d\n", a, b);       // a is 100 and b is 200
        swap(a, b);
        printf("after call a is %d and b is %d\n", a, b);        // a is 100 and b is 200
        return 0;
}
```



Version 2:

```
void swap(int *a, int *b)
{
        int temp = *a;
        *a = *b;
        *b = temp;
}
int main()
{       int a = 100;
        int b = 200;
        printf("before call a is %d and b is %d\n", a, b);      // a is 100 and b is 200
        swap(&a, &b);
        printf("after call a is %d and b is %d\n", a, b);       // a is 100 and b is 200
        return 0;
}
```



return keyword in C

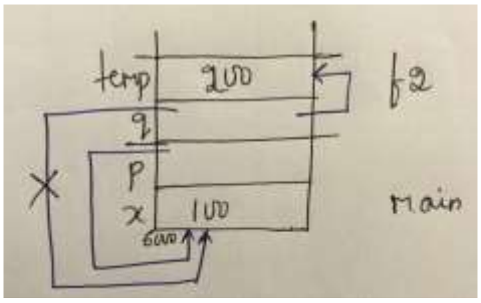
Function must do what it is supposed to do. It should not do something extra. For Example, refer to the below code.

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

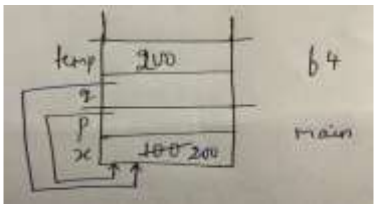
The function add() is used to add two numbers. It should not print the sum inside the function. We cannot use this kind of functions repeatedly if it does something extra than what is required. Here comes the usage of return keyword inside a function. **Syntax: return expression;**
In 'C', function returns a single value. The **expression of return is evaluated and copied to the temporary location by the called function.** This **temporary location does not have any name. If the return type of the function is not same as the expression of return in the function definition, the expression is cast to the return type before copying to the temporary location**. The calling function will pick up the value from this temporary location and use it.

Coding Example_5:

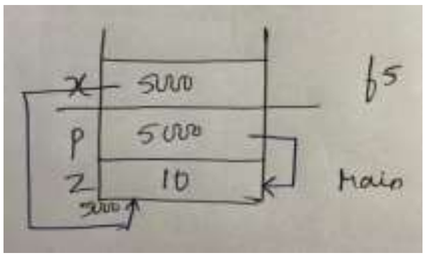
```
void f1(int);
void f2(int*);
void f3(int);
void f4(int*); i
nt* f5(int* );
int* f6();
int main()
{       int x = 100;
        f1(x);
        printf("x is %d\n", x); // 100
        double y = 6.5;
        f1(y);  // observe what happens when double value is passed as argument to integer
parameter?
        printf("y is %lf\n", y); //       6.500000
        int *p = &x; // pointer variable
        f2(p);
        printf("x is %d and *p is %d\n", x, *p);        // 100 100
        f3(*p);
        printf("x is %d and *p is %d\n", x, *p);        // 100 100
        f4(p);
        printf("x is %d and *p is %d\n", x, *p, p);
        int z= 10;
        p =f 5(&z);
        printf("z is %d and %d\n", *p, z);      // 10     10
        p = f6();
        printf("*p is %d \n", *p);
        return 0;
}

void f1(int x)
{
        x = 20;
}
void f2(int* q)
{
        int temp = 200;
        q = &temp;
}
```



```
void f3(int t)
{
        t = 200;
}
void f4(int* q)
{
        int temp = 200;
        *q = temp;
}
```

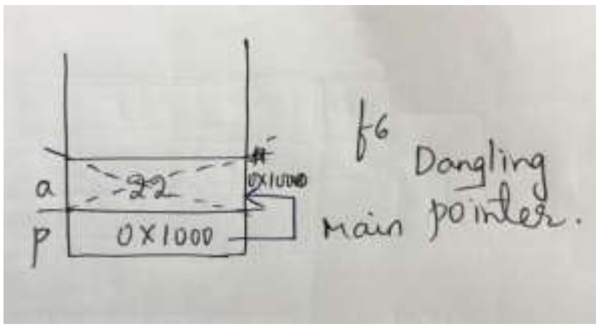


```
int* f5(int* x)
{
        return x;
}
```



When there is function call to f6, Activation Record gets created for that and deleted when the function returns. p points to a variable which is not available after the function execution. This problem is known as Dangling Pointer. The pointer is available. But the location it is not available which is pointed by pointer. Applying dereferencing operator(*) on a dangling pointer is always undefined behaviour.

```
int* f6()
{
        int a = 22;       // We should never return a pointer to a local variable
        return &a;        //          Compile time Error
}
```



When there is function call to f6, Activation Record gets created for that and deleted when the function returns. p points to a variable which is not available after the function execution. This problem is known as Dangling Pointer. The pointer is available. But the location it is not available which is pointed by pointer. Applying dereferencing operator(*) on a dangling pointer is always undefined behaviour.

void data type in c

void is an **empty data type that has no value**. We mostly use void data type in functions when we don't want to return any value to the calling function. We also use void data type in pointer like "void *p", which means, pointer "p" is not pointing to any non void data types . void * acts as generic pointer. We will be using void *, when we are not sure on the data type that this pointer will point to. We can use void * to refer either integer data or char data. void * should not be dereferenced without explicit type casting. We use void in functions as "int function_name (void)", which means, the function does not accept any argument