C Programming – Part 2

Aravind Prakash

aprakash@Binghamton.edu

Contents

- Functions and Program Structure
- Arrays
- Structures
- Unions
- Enumerators
- Macros

Contents

Function

- Has a clearly defined objective
- Accepts arguments and returns a value

return-type function-name (argument declarations)

{

declarations;

statements;

}

Function

- A declaration for a function is called the function prototype.
- A function is called from other functions or itself. When a function calls itself, it is called recursion.
- The return value of a function can either be stored into a variable, or ignored.
- ‘main’ is a special function, and is the entry point to a program.
- Parameters that are used during a function call are called ‘Actual parameters’ and those that are used in function definition and prototype are called ‘Formal parameters’

Data in the memory

- Program comprises of code and data.
- Code is usually not writable. Data may or may not be writable. Data is either:
  - In a dedicated section(s) in the memory. We must know how much space data occupies. This is the case with global data.
  - On the stack. This is the case for local variables in a function.
  - On the heap. This is the case for dynamic data (e.g., using a function like malloc).

Scope

```
#include <stdio.h>

int foo; /* Global variable, global scope. Entire program can read/write. Stored in program r/w data section. */


const int foo2 = 10; /* Global scope. Entire program can read. Stored in program read only data section */


int main()

{

        int *ptr; /* Local to main function. Stored on stack. */

        static int x; /* Local to main function, but only one copy across function calls. Stored in the data section. */

        int bar; /* Local to main function. Other functions can't access. Each entry to main function gets a new copy. Stored on the program stack. */

        ptr = malloc(1000); /* Global memory, any part of the program can access it. Stored on the heap. */

        …

}
```

Contents

- Functions and Program Structure
- Arrays
- Structures
- Unions
- Enumerators
- Macros

Arrays

- An array is an aggregate data type. It is a collection of items of data that belong to the same type.
- Array elements can be accessed by its position in the array called index.
- Index starts from 0. Last index is count-1 where count is the total number of elements in an array.
  - E.g., char c[4] stores 4 characters in index 0 through 3.
- Arrays can also be of multiple dimension.
  - E.g., float f[3][6] is a 2-dimensional array that stores 3x6=18 floating point numbers. There are accessed from a[0][0] to a[2][5].

Contents

- Functions and Program Structure
- Arrays
- Structures
- Unions
- Enumerators
- Macros

Structures

- These are user-defined types.

```
/* Definition */
struct  structure-name
{
        members;
};
/* Creating elements */
struct structure-name element;
/* Accessing members */
element.member;
```

Properties of structure

- The size of a structure is >= to sum of sizes of all its members.
- In the memory, members within a structure appear in the same order that they are declared.
  - That is, address of member_n > address of member_n-1… > address of member_1
- A structure may contain other user-defined data types within it.
- Just like regular data types, user-defined data types can be passed-to and returned-from functions.

Structure example

```
/* Definition */
struct student
{
        int b_no;
        char name[10]
        int age;
};
/* Creating elements */
struct student s;
/* Accessing members */
s.Id = 1001;
strcpy(s.name, “John”);
```

Contents

- Functions and Program Structure
- Arrays
- Structures
- Unions
- Enumerators
- Macros

Unions

- Like structures, but memory is “shared” between the members.

```
/* Definition */
union  union-name
{
        members;
};
/* Creating elements */
union union-name element;
/* Accessing members */
element.member;
```

Properties of Union

- The size of a union is equal to size of the largest member in the union.
- In the memory, at a given time, only one member of the union will hold meaningful value.
  - That is, address of member_n == address of member_n-1… == address of member_1
- A union may contain other user-defined data types within it.
- Just like regular data types, user-defined data types can be passed-to and returned-from functions.

Contents

- Functions and Program Structure
- Arrays
- Structures
- Unions
- Enumerators
- Macros

Macros

- Are for developer convenience only.
  - *#define something something-else*
  - E.g., #define TRUE 1
- During preprocessing (an early stage of compilation), *something* is ‘blindly’ replaced with *something-else*.
- As a convention, macro identifiers are in all caps.
- Macros do not have types. Think of them as search and replace.
- Make code more readable.
- Try the –E option in gcc to get the pre-processed output!

Macros example

```
#define aminusb(a, b) a-b
…
printf(“%d”, aminusb(23, 10)); /* prints 13 */
printf(“%d”, aminusb(23-10, 8-4); /* Expected 9, prints ?? Why? */
```

- Always use () to demarcate macro members.
  - #define aminusb(a, b) ((a) – (b))

Contents

- Functions and Program Structure
- Arrays
- Structures
- Unions
- Enumerators
- Macros

Enumerators

- Are integers, and can be used anywhere an integer can be used.

```
enum enum-name { members }; /* Declaration */
enum enum-name Element;  /* Creating a member */
Element.member; /*Accessing */
```

- Make code easier to read.
- Name of members is not available during runtime!

Enumerators example

```
enum Courses { cs120, cs220, cs320 };
/* Same as:
#define cs120 0
#define cs220 1
#define cs320 2
*/

enum Months { Jan=1, Feb, Mar, Nov=23, Dec};
/* Same as:
#define Jan 1
#define Feb 2
#define Mar 3
#define Nov 23
#define Dec 24 */
```