Data Structures and Algorithms (DSA) Course 3

Iulian Năstac

Recapitulation

Having the matrix :

int mat[lim_1][lim_2][lim_3]...[lim_n];

then the element ***mat[i_1][i_2]...[i_n]*** is equivalent to:

****(mat + i_1·lim_2·lim_3·...·lim_n +***
i_2·lim_3·lim_4·...·lim_n + ... + i_(n-1)·lim_n +
i_n)

Pointers (recapitulation)

- **A pointer is a variable that contains an address in the computer memory, wherein it is stored the value of another variable.**

Note: Harold Lawson is credited with the 1964 invention of the pointer.

2. Declaring pointers (recapitulation)

The declaration of a pointer type variable shall conform to the following format :

*type *name_pointer;*

As long as the declaration of a ordinary variable is ***type name***, we can say that the form ***type ** from a pointer statement is in fact a new kind of type (pointer type).

3. Operators for pointers (recapitulation)

There are different operations that can be executed with pointers, but there are two special operators which are used in unary expressions like:

operator *pointer*

These two operators are:

- ***indirection operator*** (*) → it operates on a pointer variable, and returns an l-value equivalent to the value at the pointer address.

- ***reference operator*** (&) → acts on an lvalue and the result is a pointer.

4. Expressions with pointers(recap.)

4.1. Assignment instructions for pointers

A pointer can be assigned with:

- a memory address (usually obtained with &);
- another pointer.

As observation, the format specifier used in output functions (such as **printf**) to display the value of a pointer is %p.

4.2. The arithmetic of pointers
(recapitulation)

- Arithmetic operations that can be performed using pointers are: addition (**+**) and subtraction (**-**), and supplementary, the incrementation (**++**) and decrementation (**- -**).

- **It should be noted that the pointer arithmetic is relative to their base type.**

4.3. Using a pointer with several types of data (recapitulation)

- There are cases where the same pointer can be used for multiple data types in the same program, but not simultaneously. This can be done by initially declaring the void type for the pointer:

void *name;

The Rule of Implicit conversion

It works when a binary operator is applied to two operands.

The steps of the rule :

- First convert the operands of type ***char*** and ***enum*** to the type ***int***;
- If the current operator is applied to operands of the same type then the result will be the same type. If the result is a value outside the limits of the type, then the result is wrong (exceedances occur).
- If the binary operator is applied to operands of different types, then a conversion is necessary, as in the following cases:

- If one operand is long double, therefore the other one is converted to long double and long double is the result type.

- Otherwise, if one operand is **double**, therefore the other one is converted to **double** and **double** is the result type.

- Otherwise, if one operand is **float**, therefore the other one is converted to **float** and **float** is the result type.

- Otherwise, if one operand is **unsigned long**, therefore the other one is converted to **unsigned long** and **unsigned long** is the result type.

- Otherwise, if one operand is **long**, therefore the other one is converted to **long** and **long** is the result type.

- Otherwise, if one operand is **unsigned**, therefore the other one is converted to **unsigned** and **unsigned** is the result type.

4.4. Comparing the pointers (recapitulation)

- We can compare two pointers in a relational expression.

- This comparison is however justified only if the two pointers indicate to the elements from the same array (matrix).

- Otherwise, the effect of the comparison is irrelevant as long as the compiler places the variables at different addresses depending on the available memory of the computer.

Example:

- Generate a list headed by the LIFO principle (**LIFO** – Last Input First Output). This is a stack that will store and provide integer values, according to the numbers entered.

  Thus, if you enter :

  - A value other than 0 or -1 → it is placed at the top of the stack;
  - 0 → remove a value from the stack;
  - -1 → it stops the program .

```
#include<stdio.h>
#include<stdlib.h>
#define DIMENSION 50
void push(int i);
int pop(void);
int *b, *p, stack[DIMENSION];

int main()
{       int value;
        b=stack;  /* b shows the base of the stack */
        p=stack;   /* initializes p */
        do
          {printf("\n Enter the value :");
           scanf("%d", &value);
           if(value!=0)    push(value) ;
           else printf("\n The value from the top is %d\n", pop());
         }while(value ! = -1) ;
}
```

```
void push(int i)
{
   p++;
   if(p==(b+ DIMENSION))  /* or  >= */
        { printf("\n Stack is
                        overloaded");
          exit (1);
        }
   *p = i;
}
```

```
int pop(void)
{ if(p==b)          /* or     <= */
    {printf("\n Stack is empty" );
     exit(1) ;
     }
   p - - ;
   return *(p+1) ;
}
```

5. Pointers and matrices

- The relationship between pointers and arrays has been widely discussed in the previous chapter (Matrices). C language provides two ways to access the elements of a matrix:
  - ***the arithmetic of pointers*** (the fastest way);
  - **array indices** (slower, but close to the mathematical formalism).

Remember from previous course

Having the matrix :

int mat[lim_1][lim_2][lim_3]...[lim_n];

then the element ***mat[i_1][i_2]...[i_n]*** is equivalent to:

****(mat + i_1·lim_2·lim_3·...·lim_n +***
i_2·lim_3·lim_4·...·lim_n + ... + i_(n-1)·lim_n +
i_n)

Examples:

1) In a program that contains the code:

```
...
char stg[100], *p;
p=stg;
...
```

there are similar (or equivalent) expressions of the type ***stg[4]*** and/or ****(p+4)***

Since the speed is a criterion in programming, many programmers experienced in C / C ++ usually use the pointers in order to access the array elements.

2) Two versions of a function that displays a string:

```
void write_string(char s[ ])
{       register int t;
        for(t=0; s[t]; t++)         putchar(s[t] ) ;
}
```

or:

```
void write_string(char *s)
{       while(*s)     putchar(*s++);
}
```

5.1. Matrices of pointers

Pointers can be organized in a matrix like any other data type.

Examples:

1) In a program containing matrix

int *x[10];

by using the declaration:

x[2]=&var;

the address of the variable **var** is assigned to the third element of the array of pointers.

By using

****x[2]***

we obtain the value of *var*.

2) Arrays of pointers can be used as function argument:

```
void display_matrix(int *q[ ])
{     int t;
      for(t=0; t<10; t++)
      printf("%d", *q[t]) ;
}
```

In this example, ***q*** is not a pointer to a variable of type integer, but a vector of integer pointers. Therefore it was used ****q[ ]*** in the argument of the function (to avoid any confusion).

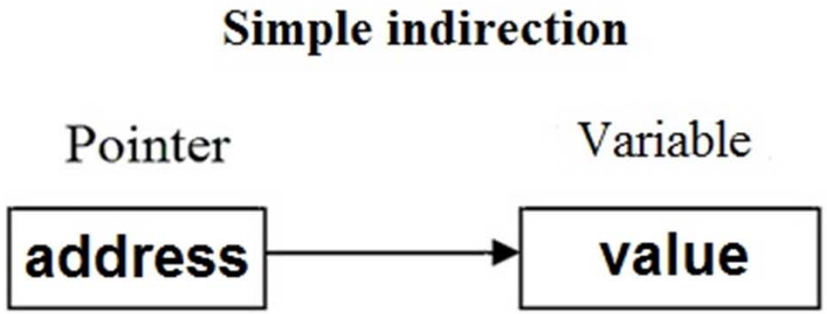
3) Arrays of pointers are often used to store pointers to the strings, as in the following example:

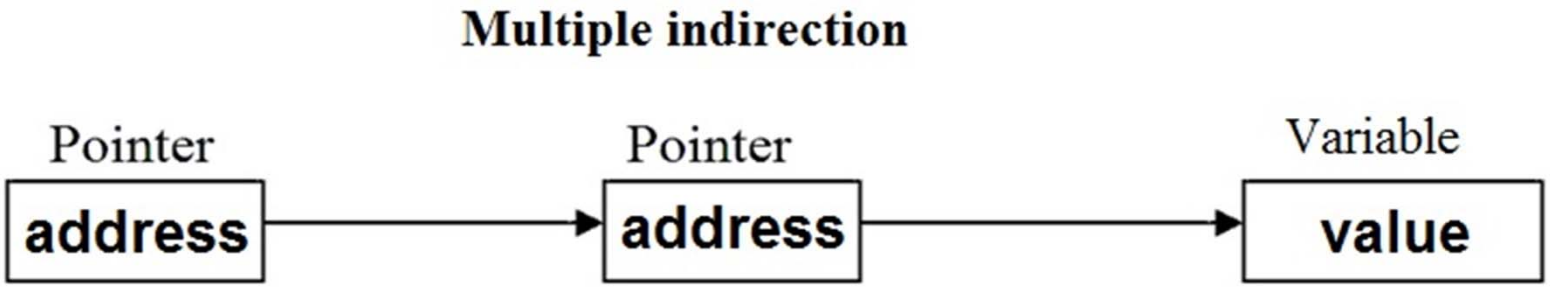
```
void error_display (int num)
{       static char *err[ ] = { " Unable to open file",
                               " Read error",
                               " Writing error",
                                ……..
                             };
        printf(" %s", err[num]) ;
}
```

The matrix ***err*** keeps pointers to each error message.

6. Multiple indirection (pointers to pointers)

- Since a pointer is itself a numeric variable, it is stored in the computer's memory at a particular address. Therefore, one can create a pointer to a pointer, a variable whose value is the address of a pointer.





Multiple indirection can be continued as long as desired (but more than a pointer to another pointer is rarely necessary).

Notes:

1) Multiple indirection must be carefully used because excessive indirection is confusing and a source of conceptual errors.
2) Multiple indirection must not be confused with high-level data structures such as chained lists, which contain pointers. The two concepts differ fundamentally.

A variable that is a pointer to another pointer is declared:

type **name ;

Example:

```
# include<stdio.h>
  int main()
     {  int x, *p, **q;
        ...
        x=10;
        p=&x;
        q=&p;
        printf(" %d", **q); /* displays the value of x */
        ...
      }
```

7. Initialize the pointers

- Once we declare a pointer, if it is not initialized, it contains an unknown value.

- A convention (in C / C ++) states that a pointer which is not initialized, has a formal null value (zero). However the fact that a pointer is null (formally), doesn't make it trusty ... up to its initialization.

Example:
A program that displays a string in both directions

```
#include<stdio.h>
#include<string.h>

char *p= ”C Programming Language”;
int main( )
    {   register  int  t;
        printf(”%s”, p); /* prints the string in a straightforward way */
        …

        for(t=strlen(p)-1; t>=0; t - -) /*inversely displayed the string*/
                printf(”%c”, p[t]);  /* virtually the same as
                                                putchar(p[t])*/
        ...
    }
```

Attention:

- To remember the specific functions for strings, we recommend to see the lecture notes in Computer Programming from:

http://www.euroqual.pub.ro/cursuri/programarea-calculatoarelor/

8. Pointers to functions

- Using pointers to functions is an important feature of the C language, but when is used incorrectly it generates confusion.

- When a program runs, the code for each function is loaded into memory starting at a specific address. A pointer to a function holds the starting address of a function (its entry point).

- The general form of the declaration is as follows:

***type* (*ptr_to_func)(*parameter_list*);**

```
#include<stdio.h>
#include<string.h>

void search(char *a, char *b, int (*comp)(const char *, const char * ));

int main()
    {   char s1[80], s2[80];
        int (*p)( const char *, const char *);
        p=strcmp;
        .....
        gets(s1);
        gets(s2);
        search(s1,s2,p);
       ....
     }

void search(char *a, char *b, int (*comp)(const char *, const char * ))
    {   printf("\n tests the equality \n" );
        if (!(*comp)(a,b)) printf("\n equal ");
        else printf("\n different");
     }
```

Remark:

- There are several functions used to compare strings. One of the most used is the **strcmp** function that has the format:

int strcmp(const char *s1, const char *s2);

Note: having two strings: **s**1 and **s**2 (*remember from previous Semester*)

- **s1 = s2** – if both have similar length and s1[i] = s2[i] ∀ i.

- **s1 < s2** – if ∃ ***i*** such that *s1[i]<s2[i]* and *s1[j]=s2[j]* ∀ j= 0, 1, …, i-1.

- **s1 > s2** – if ∃ ***i*** such that *s1[i]>s2[i]* and *s1[j]=s2[j]* ∀ j= 0, 1, …, i-1.

We can think of the ordering of words in a dictionary…

strcmp function (*remember from previous Semester*)

int strcmp(const char *s1, const char *s2);

- This function returns:
  - a negative value - if s1 < s2
  - 0 - if s1 = s2
  - a positive value - if s1 > s2

See also:

http://www.euroqual.pub.ro/cursuri/programarea-calculatoarelor/

Notes:

- In the previous example, the arguments (of the function **search(…)**) consist of two pointers to strings and **a pointer to a function**.
- It is noted that a pointer to a function introduce a significant degree of confusion (sometime without increasing the efficiency).
- The technique is still advantageous when transmitting several functions, or when we want to create a matrix of functions.
- For example instead of a large **switch** instruction with lots of features listed inside, it can be created an array of pointers to functions (making it easier to select the appropriate function).

9. Calling through the arguments of a function

(remember from previous semester)

- Calling a function in C can be realized (relative to the nature of his arguments) in two ways:

- by value;

- by reference.

10. Functions for dynamic memory allocation

- Dynamic allocation is a specific characteristic allowed by some computing languages, in which a program can obtain memory at runtime.

- There are two ways that memory gets allocated for data storage:
  - Compile Time (or static) Allocation
    - Memory for named variables is allocated by the compiler
    - Exact size and type of storage must be known at compile time
  - Dynamic Memory Allocation
    - dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
    - pointers are crucial for dynamic allocation

Notes:

- The area of memory used for dynamic allocation is obtained from the **heap** memory area.
- The C dynamic memory allocation refers to performing manual memory management for dynamic memory allocation in C programming language via a group of functions in the C standard library, namely malloc, realloc, calloc and free.
- The main functions for dynamic allocation in C are **malloc** and **free** (which can be found under header files like alloc.h or malloc.h - depending on the compiler).

malloc function allocates the specified number of bytes from the heap memory

Syntax:

*void *malloc(unsigned no_of_bytes);*

- The function returns a pointer of type void, which in fact can be assigned to any pointer.
- After a successful call, it returns a pointer to the first byte of the memory region so allocated.
- If free memory from the heap is insufficient, there is a blocking in this assignment and the function returns a NULL value.

Examples:

1) Considering a pointer to a string :

*char *p;*

one can allocate for example 1000 bytes in memory:

- in C → *p=malloc(1000);*
- in C++ → *p=(char *)malloc(1000);*

As observed in C ++, you should use a typecast.

2) Allocate space for 50 integers.

```
int *p;
p=malloc(50*sizeof(int));
```

3) A correct use of malloc requires the possibility of a lack of memory (when malloc function returns null):

```
if(!(p=malloc(100)))
{printf("Out of memory\n");
  exit(1);
}
```

free function deallocates a memory block

The released zone can be reused for a subsequent call of malloc function.

Syntax:

*void free(void *p);*

where p is a pointer to previously allocated memory.

As a remark, it should be avoid the use of the free function with an improper argument since it can destroy the running program.

Example: random string generator

```
#include <stdio.h>      /* printf, scanf, NULL */
#include <stdlib.h>     /* rand */
#include <malloc.h>     /* malloc, free */
int main ()
{  int i,n;
  char * buffer;
  printf ("How long do you want the string? ");
  scanf ("%d", &i);

  buffer = (char*) malloc (i+1);
  if (buffer==NULL) exit (1);
  for (n=0; n<i; n++)
    buffer[n]=rand()%26+'a';
  buffer[i]='\0';
  printf ("Random string: %s\n",buffer);
  free (buffer);

  return 0;
}
```

11. Incorrect use of pointers

- Often the errors provided by pointers are sometimes **difficult to detect**.
- The incorrect uses of pointers usually consist in **reading or writing in an unknown area of memory**.
- Major drawback is that the effects of these mistakes are only **visible during program execution**, and not during compilation.
- **Finding the faults is often a difficult task** for programmers, especially when, for some applications them, are not directly (or immediately) visible.

Examples:

1) Using an unknown memory location.

```
int main()
{ int x, *p;
  x=10;
  *p=x;    /* it is assigned the value of 10 to an
                 unknown location of memory */
  ...
}
```

This error may not be noticeable for small scale programs.

2) Misunderstanding of using a pointer.

```
#include<stdio.h>
int main()
{       int x, *p;
        x=10;
        p=x; /* wrong (it should be p=&x)*/

        printf("%d", *p);  /* will not show the desired
                                value of 10 */
  …
}
```

3) Incorrect assumptions about variables in the memory locations.

```
…
char s[80], y[80];
char *p1, *p2;
p1=s;
p2=y;
if(p1 < p2) … /* irrelevant task */
…
```

4) Erroneous assumption that two arrays are adjacent and can be indexed as one (by simply incrementing a pointer)

```
…
int first[10], second[10];
int *p, t;
p=first;
for(t=0; t<20; t++)     *p++ = t;

…
```

5) Omission to reset a pointer, which is allowed to cross a large area of memory. The program displays the associated ASCII values for characters in a string.

```
#include<stdio.h>
#include<string.h>
int main()
{       char *p;
        char s[80];
        p=s;
        do{     gets(s); /* read string */
                /* displays the ASCII equivalent for each character */
                while(*p)       printf(" %d ", *p++);
          } while(strcmp(s,"STOP"));
 ...
}
```

- The problem is that the pointer ***p*** is assigned to the address of ***s*** only the first time, therefore by successive incrementation it passes over other data.

The correct version is:

```
#include<stdio.h>
#include<string.h>
void main(void)
{       char *p;
        char s[80];
        do{    p=s;   /* p is reset at each new loop*/
               gets(s);
               while (*p)       printf( " %d ", *p++);
           }while(strcmp(s,"STOP"));
}
```

In conclusion, attention should be paid to the code of programs when using pointers.
It is better to anticipate the effects of their use.

Cap. Structures

1. Defining the concept of structure

- C programming language can process single or grouped variables, which enable global processing.

- An example of the second category is the matrix, which is in fact an ordered set of data of the same type (the order of the elements is realized by indices).

- However, often it is useful to group the data other than the one used for matrices. This time the data are not necessarily of the same type and requires a global processing. This form of group is called **structure**.

Reference to elements of such groups doesn't use indices but a special way that include the name of **structure**.
Components of the groups can be groups themselves. Furthermore, it is possible to define a hierarchy of such groups.

Thus:

- The group that is not part of another group is of the **highest level**;
- Data that didn't include other groups of data are **basic** (or **elementary**) data.

As a very general definition, we can say that: the data grouped according to a hierarchy are called **structures**.

Notes:

- Basic data of a structure can be isolated (single) or matrices;
  Each structure represents a new type of data, defined by the user.

2. Declaration of structure

The general syntax for a struct declaration in ***C*** is:

```
struct tag_name
    {
        type member1;
        type member2;
        …
    } identification_1, identification_2, …, identification_n;
```

Here tag_name or identification_i are optional in some contexts.

Thus:

- if identification _1, identification _2, …, identification_n are absent, then tag_name should be present.

- if tag_name is absent, then at least identification 1 should be present.

Notes:

- A variable of the structure type can be declared subsequently:

  struct tag_name identification _1, …, identification_n*;*

- A statement of a specific structure *identification_i* (where *i* = *1…n*) may be replaced by a k-dimensional array of elements of tag_name type:

  identification_i[lim1][lim2]...[limk]

Examples:

1) The following three code examples will have the same result:

```
struct calendar_data
{int day;
char month[11];
int year;
} birth_date, employment_date;
```

or

```
struct
{int day;
char month[11];
int year;
} birth_date, employment_date;
```

or

```
struct calendar_data
{int day;
char month[11];
int year;
};
...
struct calendar_data  birth_date, employment_date;
```

2) Structure containing personal information :

```
struct  personal_data
  { char name[100];
    char address[1000];
    struct calendar_data   birth_date, employment_date;
    char gender;
  };

    ..............

struct personal_data   manager, employees[1000];
```

The variable named *manager* is a structure of *personal_data* type, and *employees[1000]* is an array of structures.

3) Define complex numbers A, B and C.

```
struct COMPLEX
        {double real;
         double imag;
        }A, B, C;
```

4) The position of a point on the screen is given by two coordinates:

```
struct dot
   { int x;
     int y;
   };
   ...
struct dot position;
```

3. Access to the elements of a structure

The access to the elements of a structure can be done in one of the following two ways:

- *struct_name.date_name*
- *pointer -> date_name*

where: *struct_name is the name of* structure,
date_name is the name of a specific component of the structure,
pointer is a pointer to that structure.

Examples:

1) struct calendar_data

```
{int day;
char month[11];
int year;
} dc,d[10];
    ...
 dc.day=1;
dc.year=2015;
strcpy(dc.month,"March");
...
d[3].day=dc.day;
d[3].year=dc.year;
strcpy(d[3].month,dc.month);
...
```

2) Function that calculates and returns the ***modulus*** of the complex number z.

```
double modulus(COMPLEX *z)
        {
        return sqrt(z->x * z->x + z->y * z->y);
        }
```

It should be noted that the components of an structure can be **initialized**.

4. Typedef declarations

- By declaring a structure, we introduce a new type.
- In general, a name can be assigned to a type, whether it is a predefined type or one defined by the programmer. This should be done by using the following syntax:

typedef type new_type_name;

where

- *type* is a predefined type or one previously defined by the programmer;
- *new_type_name* is the name allocated to the new type.

Examples:

1) By using the statement

typedef double REAL;

the data

REAL x,y;

are of the double type.

2) Declaring COMPLEX type.

```
typedef struct
        { double real;
          double imag;
        } COMPLEX;
...
```

We can then declare complex numbers:

COMPLEX z, tz[10];

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<conio.h>
typedef struct {
                double x;
                double y;
               } COMPLEX;
void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c);

int main()
{
 COMPLEX a,b,c;
 printf("\n\n Enter the real and the imaginary part ");
 printf("\n of the first complex number :\n");
 if(scanf("%lf %lf",&a.x,&a.y)!=2)
        {
          printf("\nError");
          exit(1);
         }
 ….
```

```
…
printf("a = %g + i*(%g)\n",a.x,a.y);
 printf("\n\n Enter the real and the imaginary part ");
 printf("\n of the second complex number :\n");
 if(scanf("%lf %lf",&b.x,&b.y)!=2)
        {
          printf("\nError");
          exit(1);
         }
 printf("b = %g + i*(%g)\n",b.x,b.y);
 sum_c(&a,&b,&c);
 printf("\na+b = %g + i*(%g)",c.x, c.y);

 getch();
}


void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c)
{
  c->x = a->x + b->x;
  c->y = a->y + b->y;
}
```

5. Unions

Introduction

Usually, in C a memory area is assigned according to the type of variable. Its allocated memory can keep only the data of the mentioned type.

For example:

double x;

For **x** is allocated 8 bytes (64 bits) in the computer memory in order to store a real number.

What is an union in C?

- Unions in C are related to structures and are defined as objects that may hold (at different times) objects of different types and sizes.
- They are analogous to variant records in other programming languages. Unlike structures, the components of a union all refer to the same location in memory.
- In this way, a union can be used at various times to hold different types of objects, without the need to create a separate object for each new type.
- The size of a union is equal to the size of its largest component type.

Definition

- A **union** is a special data type available in C that enables you to store different data types in the same memory location.

- Notes:
  - You can define a union with many members, but only one member can contain a value at any given time.
  - Unions provide an efficient way of using the same memory location for multi-purpose.

Examples:

1)
```
union a
        {int x;         /* 2 bytes for x */
        long y;         /* 4 bytes for y */
        double r;       /* 8 bytes for r*/
        char c;         /* 1 byte for c */
        } var;
```

In the above statement ***var*** is a union of the type **a**. Accessing variables can be done with: *var.x;* or *var.y;* or *var.r;* or *var.c*; but in different locations of the program

For ***var*** it is allocated a memory area which is sufficient to keep the maximum number of bytes (8 bytes in this example). If ***union*** would be replaced with ***struct***, then 15 bytes would be required (2+4+8+1=15).

2)

```
struct data
        { int timp;
          union { int i;
                  float f;
                  double d;
                } zc;
        } util;
```

We can access:

```
util.zc.i=123;
```

As observed, in contrast to the structure, a union can not be initialized.

6. Bit fields

- C also provides a special type of structure member known as a **bit field**, which is an integer with an explicitly specified number of bits.

- A bit field is declared as a structure member of type int, signed int, unsigned int, or boolean, following the member name by a colon (:) and the number of bits it should occupy.

- The total number of bits in a single bit field must not exceed the total number of bits in its declared type.

Basically, more fields can be grouped to form a structure:

```
struct identification
    { field_1;
      field_2;
     ...
      field_n;
    } name_1, name_2, ..., name_n;
```

Syntax of bit field:

type name_field: length_in_bits;

or:

type : length_in_bits;

where type can be int, signed int, unsigned int, or boolean.

Notes:

- As a special exception to the usual C syntax rules, it is implementation-defined whether a bit field declared as type **int**, without specifying signed or unsigned, is signed or unsigned. Thus, it is recommended to explicitly specify signed or unsigned on all structure members for portability.

- Unnamed fields consisting of just a colon followed by a number of bits are also allowed; these indicate **padding**. Specifying a width of zero for an unnamed field is used to force alignment to a new word.