

Structures, Operators

Basics of Programming 1



G. Horváth, A.B. Nagy, Z. Zsóka, P. Fiala, A. Vitéz

08 October, 2025

Content

1 Structures

- Motivation
- Definition
- Assignment of value

2 Typename-assignment

3 Operators

- Definitions

- Operators

- Precedence

4 The enumerated type

- Motivation

- Syntax

- Examples

Chapter 1

Structures

User defined types

Built-in types of C language sometimes are not appropriate for storing more complex data.

Types introduced by the user (programmer)

- Enumeration
- Structures ← today's topic
- Bitfields
- Union

Data elements that are coupled

■ Storing date

```

1  int year;
2  int month;
3  int day;

```

■ Storing student data

```

1  char neptun[6];
2  unsigned int smallestests;
3  unsigned int missings;

```



- Data of a chess game
(white player, black player, when, where, **moves**, result)
- Data of one move
(chess piece, **from where**, where to)
- Data of one square of the board
(column, row)

Storing data elements that are coupled

- Let's write a function to calculate scalar product (dot product) of 2D vectors!

```
1 double v_scalarproduct(double x1, double y1,  
2                       double x2, double y2)  
3 {  
4     ...  
5 }
```

How shall we pass coupled parameters?

The number of parameters may become too large

- Let's write a function to calculate difference of two vectors!

```
1 ?????? v_difference(double x1, double y1,  
2                   double x2, double y2)  
3 {  
4     ...  
5 }
```

How does the function returns with coupled data?

Encapsulation

Structure

compound data type consisting of data elements (maybe of different types) that are coupled (belong together)

student

neptun
small test results
missings

- data elements are called fields or members
- can be copied with one assignment
- can be parameter of function
- can be return value of function
- This is the most effective of C language

Structures in C

```
1  struct vector { /* definition of structure type */
2     double x; double y;
3 };
4
5  struct vector v_difference(struct vector a,
6                             struct vector b) {
7     struct vector c;
8     c.x = a.x - b.x;
9     c.y = a.y - b.y;
10    return c;
11 }
12
13 int main(void) {
14     struct vector v1, v2, v3;
15     v1.x = 1.0; v1.y = 2.0;
16     v2 = v1;
17     v3 = v_difference(v1, v2);
18     return 0;
19 }
```


Syntax of structures

Declaration of structures

```
struct [<structure label>]opt  
{<structure member declarations>}  
[<variable identifiers>]opt;
```

```
1 /* structure type for storing date */  
2 struct date {  
3     int year;  
4     int month;  
5     int day;  
6 } d1, d2; /* two instances (variables) */
```

- [<structure label>]_{opt}
can be omitted if we don't refer to it later
- [<variable identifiers>]_{opt}
declaration of variables of structure type

Syntax of structures

Using structure type

- Declaration of variables

```
struct <structure label> <variable identifiers>;
```

- Accessing structure members

```
<structure identifier>.<member identifier>
```

- Structure members can be used in the same way as variables

```
1 struct date d1, d2;  
2 d1.year = 2012;  
3 d2.year = d1.year;  
4 scanf("%d", &d2.month);
```

- Initialization of structures is possible in the same way as for arrays:

```
1 struct date d3 = {2011, 5, 2};
```

Assignment of value to structures

- Value of a structure variable (value of all members) can be updated with **one single** assignment.

```
1 struct date d3 = {2013, 10, 22}, d4;  
2 d4 = d3;
```

Chapter 2

Typename-assignment

Definition

- We can rename types in C

```
1 typedef int rabbit;  
2  
3 rabbit main() {  
4     rabbit i = 3;  
5     return i;  
6 }
```

Typename-assignment

- `typedef` assigns a nickname to the type.
- It does not create a new type, the type of all variables created with the nickname will be the original type.

What is the use of it?

- More meaningful source code, more easy to read

```
1 typedef float voltage; /* we need a smaller */  
2  
3 voltage V1 = 1.0;  
4 double c = 2.0;  
5 voltage V2 = c * V1;
```

- Easy to maintain
- We can get rid of typenames of more than one word

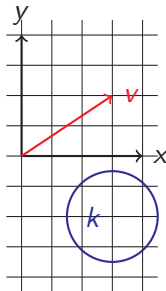
```
1 typedef struct vector vector;
```

Vector example with typedef

```
1 typedef struct { /* we can omit the label */
2     double x; double y;
3 } vector;
4
5
6 vector v_difference(vector a, vector b) {
7     vector c;
8     c.x = a.x - b.x;
9     c.y = a.y - b.y;
10    return c;
11 }
12
13 int main(void) {
14     vector v1, v2, v3;
15     v1.x = 1.0; v1.y = 2.0;
16     v2 = v1;
17     v3 = v_difference(v1, v2);
18     return 0;
19 }
```

A more complex structure

```
1 typedef struct {
2     double x;
3     double y;
4 } vector;
5
6 typedef struct {
7     vector centrepoint;
8     double radius;
9 } circle;
10
11 circle k = {{3.0, 2.0}, 1.5};
12 vector v = k.centrepoint;
13 k.centrepoint.y = -2.0;
```

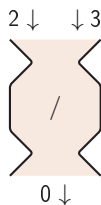
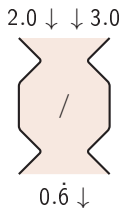


Chapter 3

Operators

Operations

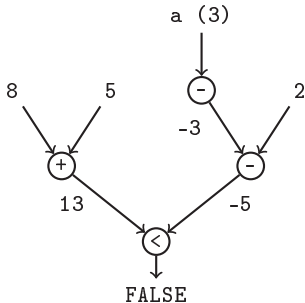
- Denoted with operators (special symbols)
- They work with operands
- They result a data with type
- Polymorphic: have different behaviour on different operand types



Expressions and operators

■ Expressions

- eg. $8 + 5 < -a - 2$
- Built up of constants, variable references and operations



- by evaluating them the result is one data element with type.

Types of operators

- Considering the number of operands
 - unary – with one operand
 - a
 - binary – with two operands
 - 1+2
- Considering the interpretation of the operand
 - arithmetic
 - relational
 - logical
 - bitwise
 - misc

Arithmetic operators

operation	syntax
unary plus	<code>+<expression></code>
unary minus	<code>-<expression></code>
addition	<code><expression> + <expression></code>
subtraction	<code><expression> - <expression></code>
multiplication	<code><expression> * <expression></code>
division	<code><expression> / <expression></code>
type of the result depends on type of the operands, if both are integer, then it is an integer division	
modulus	<code><expression> % <expression></code>

True or false – Boolean in C (repeated)

- Every boolean like result is `int` type, and its value is
 - 0, if false
 - 1, if true

```
1 printf("%d\t%d", 2<3, 2==3);
```

```
1 0
```

- A value interpreted as boolean is
 - false, if its value is represented with 0 bits only
 - true, if its value is represented with **not** only 0 bits

```
1 while (1)    { /* infinite loop */ }  
2 while (-3.0) { /* infinite loop */ }  
3 while (0)    { /* this here is never executed */ }
```

Relational operators

operation	syntax
relational operators	<code><left value> < <expression></code>
	<code><left value> <= <expression></code>
	<code><left value> > <expression></code>
	<code><left value> >= <expression></code>
checking equality	<code><left value> == <expression></code>
checking non-equality	<code><left value> != <expression></code>

They give logical value (`int`, 0 or 1) as result.

Logical operators

operation

syntax

logical NOT (complement) !<expression>

```
1 int a = 0x5c; /* 0101 1100, true */
2 int b = !a;   /* 0000 0000, false */
3 int c = !b;   /* 0000 0001, true  */
```

- Conclusion: $!!a \neq a$, only if we look at their boolean value.

```
1 int finish = 0;
2 while (!finish) {
3     int b;
4     scanf("%d", &b);
5     if (b == 0)
6         finish = 1;
7 }
```


Logical operators

operation	syntax
logical AND	<expression> <code>&&</code> <expression>
logical OR	<expression> <code> </code> <expression>

Logical short-cut: Operands are evaluated from left to right. But only until the result is not obvious.

We make use of this feature very often.

```
1 int a[5] = {1, 2, 3, 4, 5};
2 int i = 0;
3 while (i < 5 && a[i] < 20)
4     i = i+1; /* no over-indexing */
```

Some more operators

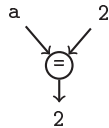
We have used them so far, but never have called them operators before.

operation	syntax
function call	<function>(<actual arguments>)
array reference	<array> [<index>]
structure-reference	<structure>.<member>

```
1 c = sin(3.2); /* () */
2 a[28] = 3; /* [] */
3 v.x = 2.0; /* . */
```

Operators with side effects

- Some operators have side effects
 - main effect: calculating the result of evaluation
 - side effect: the value of the operand is modified
- Simple assignment operator =
 - In C language, assignment is an expression!
 - its side effect is the assignment (a is modified)
 - its main effect is the new value of a
 - Because of its main effect, this is also meaningful:



```
1 int a;  
2 int b = a = 2;
```

- b is initialised with the value of expression a=2 (this also has a side effect), and the side effect of it is that a is also modified.

Left-value

- Assignment operator modifies value of the left side operand. There can be only "modifiable entity" on the left side.

Left-value (lvalue)

An expression that can appear on the left side of the assignment.

- As far as we know now, left-value can be
 - a variable reference `a` = 2
 - element of an array `array[3]` = 2
 - member of a structure `v.x` = 2
 - ...
- Examples for non-left-value expressions
 - constant `3` = 2 error
 - arithmetic expression `a+4` = 2 error
 - logical expression `a>3` = 2 error
 - function value `sin(2.0)` = 2 error

Expression or statement?

An operation that has side effect can be a statement in a program.

Expression statement

```
<Expression>;
```

- Expression is evaluated, but the result is thrown away (but all side effects are completed).

```
1 a = 2 /* expression, its value is 2, it has side effect */
```

```
1 a = 2; /* statement, it has no value */  
2 /* generates a side effect */
```

- As the main effect is suppressed, there is no sense of making expression statements if the expression has no side effect.

```
1 2 + 3; /* valid statement, it generates nothing */
```

Assignment operators

expression	syntax
	<code><left-value> += <expression></code>
	<code><left-value> -= <expression></code>
compound assignment	<code><left-value> *= <expression></code>
	<code><left-value> /= <expression></code>
	<code><left-value> %= <expression></code>

■ **Almost:** `<left-value>=<left-value><op><expression>`

```

1 a += 2;          /* a = a + 2; */
2 t[rand()] += 2; /* NOT t[rand()] = t[rand()] + 2; */

```

Left-value is evaluated only once.

Other operators with side effects

expression

syntax

post increment <left-value> ++

post decrement <left-value> --

it is increased/decreased by one **after** evaluation

pre increment ++<left-value>

pre decrement --<left-value>

it is increased/decreased by one **before** evaluation

```
1 b = a++; /* b = a; a += 1; */
2 b = ++a; /* a += 1; b = a; */
```

```
1 for (i = 0; i < 5; ++i) { /* five times */ }
```

Other operators

operation

syntax

modifying type
(casting)

`(<type>)<expression>`

size for storage (in bytes)
the expression is not evaluated

`sizeof <expression>`

```
1 int a1=2, a2=3, storagesize;  
2 double b;  
3 b = a1/(double)a2;  
4 storagesize = sizeof 3/a1;  
5 storagesize = sizeof(double)a1;  
6 storagesize = sizeof(double);
```


Other operators

operation	syntax
comma	<expression> , <expression>

- Operands are evaluated from left to right.
- Value of first expression is thrown away.
- Value and type of the entire expression is the value and type of the second expression.

```
1 int step, j;
2 /* two-digit numbers with increasing step size */
3 for(step=1, j=10; j<100; j+=step, step++)
4     printf("%d\n", j);
```

Other operators

operation

syntax

(ternary) conditional expr. `<cond.> ? <expr.1> : <expr.2>`

- if `<cond.>` is true, then `<expr.1>`, otherwise `<expr.2>`.
- only one of `<expr.1>` and `<expr.2>` is evaluated.
- It does not substitute the `if` statement.

```
1 a = a < 0 ? -a : a; /* determining absolute value */
```

Features of operations performed on data

Precedence

If there are **different** operations, which is evaluated first?

```
1 int a = 2 + 3 * 4; /* 2 + (3 * 4) */
```

Associativity

If there are **equivalent** operations, which is evaluated first?
(Does it bind from left to right or from right to left?)

```
1 int b = 11 - 8 - 2; /* (11 - 8) - 2 */
```

Instead of memorizing the rules, use parentheses!

List of operators in C

Operators are listed top to bottom, in descending precedence (operators in the same row have the same precedence)

```

1  ()  []  .  -> /* highest */
2  !  ~  ++  --  +  -  *  &  (<type>)  sizeof
3  *  /  %
4  +  -
5  <<  >>
6  <  <=  >  >=
7  ==  !=          /* forbidden to learn! */
8  &          /* use parentheses! */
9  ^
10 |
11 &&
12 ||
13 ?:
14 =  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=
15 , /* lowest */

```

Operators of C language

Summarized

- A lot of effective operators
- Some operators have side effects that will occur during evaluation
- We always try to separate main and side effects
Instead of this:

```
1 t[++i] = func(c-=2);
```

we rather write this:

```
1 c -= 2;          /* means the same */
2 ++i;           /* not less effective */
3 t[i] = func(c); /* and I will understand it tomorrow too */
```

Chapter 4

The enumerated type

The enumerated type – Motivation

- We are writing a game, in which the user can control direction of the player with 4 keys.



- As the input from user needs to be read (checked) frequently, we create a `read_direction()` function for this task.
- This function reads from the keyboard and returns the direction to the calling program segment.
- What type should the function return with?

The enumerated type – Motivation

- Idea Nr. 1: Let's return with the key pressed.
(`'a'`, `'s'`, `'w'`, `'d'`):

```
1 char read_direction(void)
2 {
3     char ch;
4     scanf("%c", &ch);
5     return ch;
6 }
```

[link](#)

- Problems:
 - We have to decode characters into directions many times at different parts of the source code.
 - If we change to use the arrow keys \leftarrow \downarrow \uparrow \rightarrow for control, we have to modify the source code a thousand time and place.
- Solution:
 - We have to decode in place (inside the function), and should return with direction.
 - But how can we do that?

The enumerated type – Motivation

- Idea Nr. 2: Let's return with `int` values 0,1,2,3:

```
'a'  0  ← 1  int read_direction(void) {  
      2      char ch;  
'w'  1  ↑ 3      scanf("%c", &ch);  
      4      switch (ch) {  
'd'  2  → 5      case 'a': return 0; /* left */  
      6      case 'w': return 1; /* up */  
's'  3  ↓ 7      case 'd': return 2; /* right */  
      8      case 's': return 3; /* down */  
      9      }  
     10     return 0; /* default is left :) */  
     11 }
```

- Problem:
 - In other parts of the program we have to use numbers 0-3 for the directions, so the programmer **must remember** the number-direction assignments.

The enumerated type – Motivation

- We need a type named `direction`, that can store `LEFT`, `RIGHT`, `UP`, `DOWN` values.
- We can do such thing in C!

Declaration of the appropriate enumerated type (`enum`):

```
1 enum direction {LEFT, RIGHT, UP, DOWN};
```

- How to use the type:

```
1 enum direction d;  
2 d = LEFT;
```

The enumerated type – Motivation

■ The final solution with the new type

```
1 enum direction {LEFT, RIGHT, UP, DOWN};
2
3
4 enum direction read_direction(void)
5 {
6     char ch;
7     scanf("%c", &ch);
8     switch (ch)
9     {
10    case 'a': return LEFT;
11    case 'w': return UP;
12    case 'd': return RIGHT;
13    case 's': return DOWN;
14    }
15    return LEFT;
16 }
```

[link](#)

The enumerated type – Motivation

■ Usage of the function:

```
1 enum direction d = read_direction();  
2 if (d == RIGHT)  
3     printf("You were eaten by a tiger\n"); link
```

■ Without the enumerated type, it would look like this:

```
1 int d = read_direction();  
2 if (d == 2) /* "magic" constant, what does it mean? */  
3     printf("You were eaten by a tiger\n"); link
```

■ The enumerated type...

- replaces "magic constants" with informative code,
- focuses on content instead of representation,
- allows a higher level programming.

The enumerated type – Definition

The enumerated (enum) type

Joins into one type integer type constants referenced by symbolic names.

```
enum [<enumeration label>]opt  
{ <enumeration list> }  
[<variable identifiers>]opt;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

enum examples

```
1 enum month {
2     JAN, /* 0 */
3     FEB, /* 1 */
4     MAR, /* 2 */
5     APR, /* 3 */
6     MAY, /* 4 */
7     JUNE, /* 5 */
8     JULY, /* 6 */
9     AUG, /* 7 */
10    SEPT, /* 8 */
11    OCT, /* 9 */
12    NOV, /* 10 */
13    DEC /* 11 */
14 };
15
16 enum month m=OCT; /*9*/
```

```
1 enum {
2     RED, /* 0 */
3     BLUE = 3, /* 3 */
4     GREEN, /* 4 */
5     YELLOW, /* 5 */
6     GRAY = 10 /* 10 */
7 } c;
8
9 c = GREEN;
10 printf("c: %d\n", c);
```

```
c: 4
```

Thank you for your attention.