Structures, Operators

Basics of Programming 1

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

9 October, 2024

Content

Chapter 1

Structures

User defined types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

Types introduced by the user (programmer)

- Enumeration
- Structures
- Bitfields
- Union

User defined types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Storing date

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

Data elements that are coupled

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Storing date

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

- Storing student data

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

Data elements that are coupled

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Storing date

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

- Storing student data

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

- Data of a chess game (white player, black player, when, where, moves, result)

Data elements that are coupled

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Storing date

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

- Storing student data

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

- Data of a chess game (white player, black player, when, where, moves, result)

Data elements that are coupled

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Storing date

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

- Storing student data

```
1  char neptun[6];
2  unsigned int smalltests;
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 elements that are coupled

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Storing date

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

- Storing student data

```
1  char neptun[6];
2  unsigned int smalltests;
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 elements that are coupled

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Storing date

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

- Storing student data

```
1  char neptun[6];
2  unsigned int smalltests;
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

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!

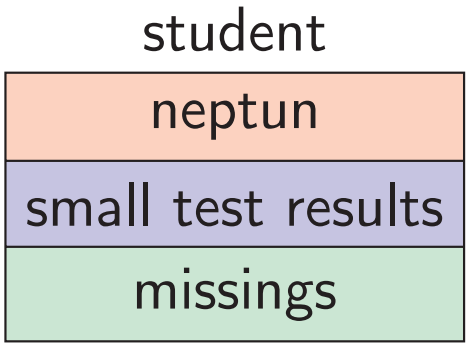
```
1  ?????? v_difference(double x1, double y1,
2                      double x2, double y2)
3  {
4  ...
5  }
```

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)



- 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 type of C language

Structures in C

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
struct vector { /* definition of structure type  */
  double x; double y;
};

struct vector v_difference(struct vector a,
                           struct vector b) {
  struct vector c;
  c.x = a.x - b.x;
  c.y = a.y - b.y;
  return c;
}

int main(void) {
  struct vector v1, v2, v3;
  v1.x = 1.0;  v1.y = 2.0;
  v2 = v1;
  v3 = v_difference(v1, v2);
  return 0;
}
```

Syntax of structures

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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) */
```

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) */
```

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) */
```

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) */
```

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

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>;`

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

Syntax of structures

Using structure type

- Declaration of variables

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

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

Syntax of structures

Using structure type

- Declaration of variables

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

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

Syntax of structures

Using structure type

- Declaration of variables

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

- Accessing structure members

  `<structure identifier>.<member identifier>`

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

Syntax of structures

Using structure type

- Declaration of variables

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

- Accessing structure members

  `<structure identifier>.<member identifier>`

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

Syntax of structures

Using structure type

- Declaration of variables

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

- Accessing structure members

  `<structure identifier>.<member identifier>`

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

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);
```

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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  }
```

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 nicname will be the original type.

What is the use of it?

- More meaningful source code, more easy to read

```
1 typedef double voltage;
2
3 voltage V1 = 1.0;
4 double c = 2.0;
5 voltage V2 = c * V1;
```

What is the use of it?

- More meaningful source code, more easy to read

```
1 typedef long double voltage; /* we need more accuracy */
2
3 voltage V1 = 1.0;
4 double c = 2.0;
5 voltage V2 = c * V1;
```

- Easy to maintain

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

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
struct vector { /* new structure type */
  double x; double y;
};
typedef  struct vector  vector; /* renaming */

vector v_difference(vector a, vector b) {
  vector c;
  c.x = a.x - b.x;
  c.y = a.y - b.y;
  return c;
}

int main(void) {
  vector v1, v2, v3;
  v1.x = 1.0;  v1.y = 2.0;
  v2 = v1;
  v3 = v_difference(v1, v2);
  return 0;
}
```

Vector example with typedef

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
typedef struct vector { /* done in one step */
  double x; double y;
} vector;


vector v_difference(vector a, vector b) {
  vector c;
  c.x = a.x - b.x;
  c.y = a.y - b.y;
  return c;
}

int main(void) {
  vector v1, v2, v3;
  v1.x = 1.0;  v1.y = 2.0;
  v2 = v1;
  v3 = v_difference(v1, v2);
  return 0;
}
```

Vector example with typedef

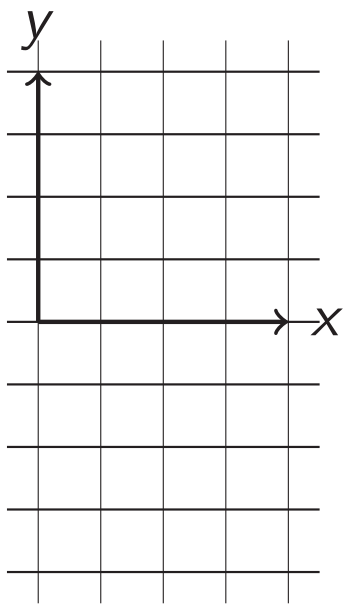
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
typedef struct { /* we can omit the label */
  double x; double y;
} vector;


vector v_difference(vector a, vector b) {
  vector c;
  c.x = a.x - b.x;
  c.y = a.y - b.y;
  return c;
}

int main(void) {
  vector v1, v2, v3;
  v1.x = 1.0;  v1.y = 2.0;
  v2 = v1;
  v3 = v_difference(v1, v2);
  return 0;
}
```

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;
```

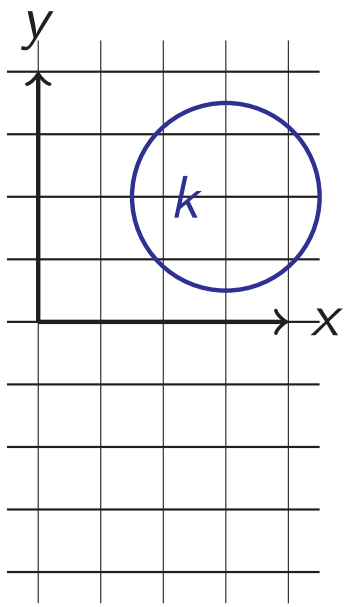
```
1  circle k = {{3.0, 2.0}, 1.5};
2  vector v = k.centrepoint;
3  k.centrepoint.y = -2.0;
```



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;
```

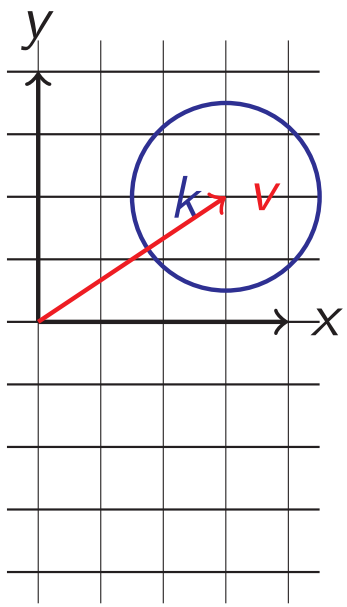
```
1  circle k = {{3.0, 2.0}, 1.5};
2  vector v = k.centrepoint;
3  k.centrepoint.y = -2.0;
```



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;
```

```
1  circle k = {{3.0, 2.0}, 1.5};
2  vector v = k.centrepoint;
3  k.centrepoint.y = -2.0;
```

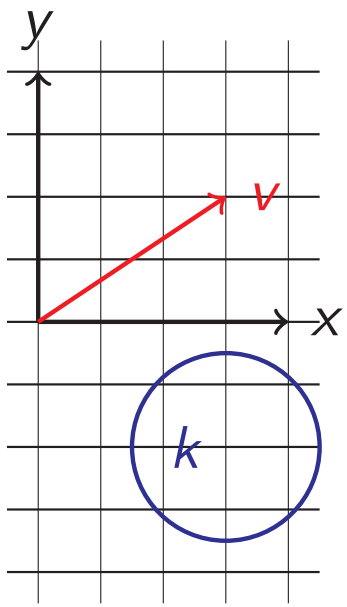


A more complex structure

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  typedef struct {
2    double x;
3    double y;
4  } vector;
5
6  typedef struct {
7    vector centrepoint;
8    double radius;
9  } circle;
```

```
1  circle k = {{3.0, 2.0}, 1.5};
2  vector v = k.centrepoint;
3  k.centrepoint.y = -2.0;
```

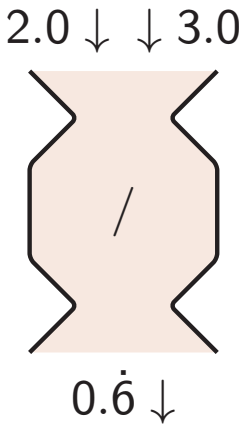


Chapter 3

Operators

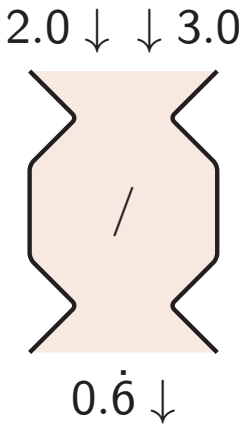
Operations

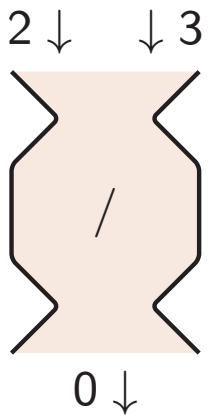
- Denoted with operators (special symbols)
- They work with operands
- They result a data with type



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

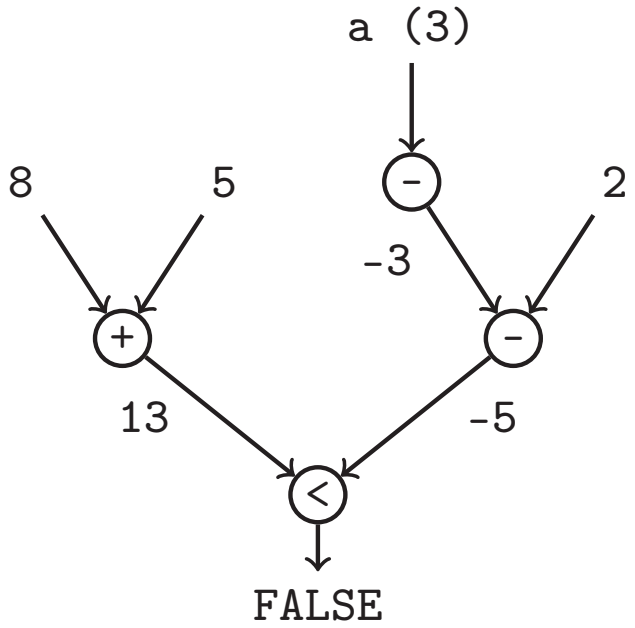
Expressions and operators

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Expressions
  - eg. `8 + 5 < -a - 2`

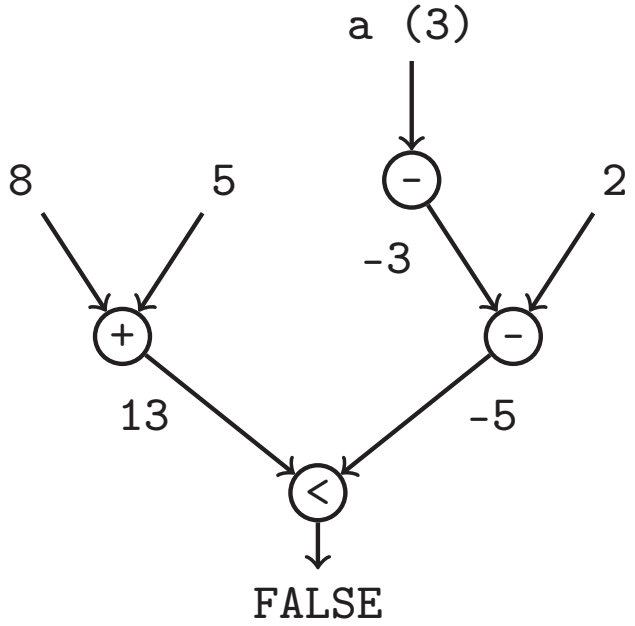
Expressions and operators

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



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

Types of operators

- Considering the number of operands
  - unary – with one operand

    `-a`

Types of operators

- Considering the number of operands
  - unary – with one operand
    `-a`
  - binary – with two operands
    `1+2`

Types of operators

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Considering the number of operands
  - unary – with one operand
    `-a`
  - binary – with two operands
    `1+2`
- Considering the interpretation of the operand

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

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

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

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

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

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
```

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	`<left value> < <expression>`
	`<left value> <= <expression>`
	`<left value> > <expression>`
	`<left value> >= <expression>`
checking equality	`<left value> == <expression>`
checking non-equality	`<left value> != <expression>`

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 */
```

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

Logical operators

operation	syntax
logical NOT (complement)	`!<expression>`

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

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

```
int finish = 0;
while (!finish) {
  int b;
  scanf("%d", &b);
  if (b == 0)
    finish = 1;
}
```

Logical operators

operation	syntax
logical AND	`<expression> && <expression>`
logical OR	`<expression> || <expression>`

Logical operators

operation	syntax
logical AND	`<expression>` && `<expression>`
logical OR	`<expression>` || `<expression>`

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

Logical operators

operation	syntax
logical AND	`<expression> && <expression>`
logical OR	`<expression> || <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>`

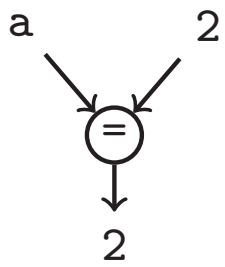
```
1  c = sin(3.2); /* () */
2  a[28] = 3;    /* [] */
3  v.x = 2.0;    /* . */
```

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

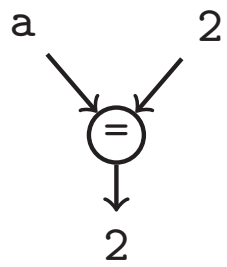
Operators with side effects

- 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



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

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

Left-value

- Assignement 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.

Left-value

- Assignement 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`
  - ...

Left-value

- Assignement 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 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).

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 */
```

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 */
```

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 surpressed, there is no sense of making expression statements if the expression has no side effect.

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

Assignement operators

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

Assignement operators

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

- 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>`

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
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
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
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
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.

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
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 subtitute the `if` statement.

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

Features of operations performed on data

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Precedence

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

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

Features of operations performed on data

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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 */
```

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

Operateors 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

Operators of C language

Summarized

- A lot of effective operators
- Some operators have side effects that will occur during evaluation

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 */
```

Thank you for your attention.