

Operators – Pointers

Basics of Programming 1



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

21 October, 2020

Content

1 Operators

- Definitions
- Operators
- Precedence

2 Type conversion

3 Pointers

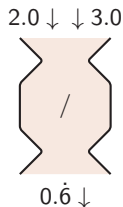
- Definition of pointers
- Passing parameters as address
- Pointer-arithmetics
- Pointers and arrays

Chapter 1

Operators

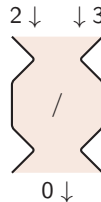
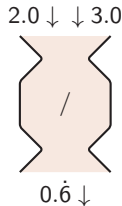
Operations

- 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

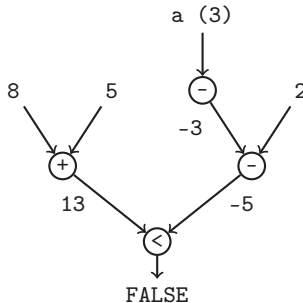
- Expressions

- eg. $8 + 5 < -a - 2$

Expressions and operators

■ Expressions

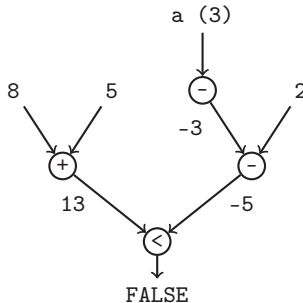
- eg. $8 + 5 < -a - 2$
- Built up of constants, variable references and operations



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

- 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	$+\langle\text{expression}\rangle$
unary minus	$-\langle\text{expression}\rangle$
addition	$\langle\text{expression}\rangle + \langle\text{expression}\rangle$
subtraction	$\langle\text{expression}\rangle - \langle\text{expression}\rangle$
multiplication	$\langle\text{expression}\rangle * \langle\text{expression}\rangle$
division	$\langle\text{expression}\rangle / \langle\text{expression}\rangle$
type of the result depends on type of the operands, if both are integer, then it is an integer division	
modulus	$\langle\text{expression}\rangle \% \langle\text{expression}\rangle$

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	<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.

Logical operators

operation	syntax
logical NOT (complement)	<code>!<expression></code>

```
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> && <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	<code><function>(<actual arguments>)</code>
array reference	<code><array>[<index>]</code>
structure-reference	<code><structure>.<member></code>

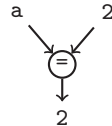
```
1 c = sin(3.2); /* () */
2 a[28] = 3;    /* [] */
3 v.x = 2.0;    /* .  */
```

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

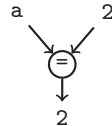
Operators with side effects

- 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

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

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.

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
 - ...

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 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 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>

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)	<code>(<type>)<expression></code>
size for storage (in bytes) the expression is not evaluated	<code>sizeof <expression></code>

Other operators

operation	syntax
modifying type (casting)	<code>(<type>)<expression></code>
size for storage (in bytes) the expression is not evaluated	<code>sizeof <expression></code>

```
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)	<code>(<type>)<expression></code>
size for storage (in bytes) the expression is not evaluated	<code>sizeof <expression></code>

```
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)	<code>(<type>)<expression></code>
size for storage (in bytes) the expression is not evaluated	<code>sizeof <expression></code>

```
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)	<code>(<type>)<expression></code>
size for storage (in bytes) the expression is not evaluated	<code>sizeof <expression></code>

```
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)	<code>(<type>)<expression></code>
size for storage (in bytes) the expression is not evaluated	<code>sizeof <expression></code>

```
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	<code><expression> , <expression></code>

- 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 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) */
```

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 */
```

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

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 */
```

Chapter 2

Type conversion

What is that?

In some cases the C-program needs to convert the type of our expressions.

```
1 long func(float f) {  
2     return f;  
3 }  
4  
5 int main(void) {  
6     int i = 2;  
7     short s = func(i);  
8     return 0;  
9 }
```

In this example: `int` \rightarrow `float` \rightarrow `long` \rightarrow `short`

- `int` \rightarrow `float` rounding, if the number is large
- `float` \rightarrow `long` may cause overflow, rounding to integer
- `long` \rightarrow `short` may cause overflow

Converting types

- Basic principle

Converting types

- Basic principle
 - preserve the value, if possible

Converting types

- Basic principle
 - preserve the value, if possible
- In case of overflow

Converting types

- Basic principle
 - preserve the value, if possible
- In case of overflow
 - the result is theoretically undefined

Converting types

- Basic principle
 - preserve the value, if possible
- In case of overflow
 - the result is theoretically undefined
- Conversion with one operand (we have seen that)

Converting types

- Basic principle
 - preserve the value, if possible
- In case of overflow
 - the result is theoretically undefined
- Conversion with one operand (we have seen that)
 - at assignment of value

Converting types

- Basic principle
 - preserve the value, if possible
- In case of overflow
 - the result is theoretically undefined
- Conversion with one operand (we have seen that)
 - at assignment of value
 - at calling a function (when actualising the formal parameters)

Converting types

- Basic principle
 - preserve the value, if possible
- In case of overflow
 - the result is theoretically undefined
- Conversion with one operand (we have seen that)
 - at assignment of value
 - at calling a function (when actualising the formal parameters)
- Conversion with two operands (eg. $2/3.4$)

Converting types

- Basic principle
 - preserve the value, if possible
- In case of overflow
 - the result is theoretically undefined
- Conversion with one operand (we have seen that)
 - at assignment of value
 - at calling a function (when actualising the formal parameters)
- Conversion with two operands (eg. $2/3.4$)
 - evaluating an operation

Conversion with two operands

The conversion of the two operands to the same, common type happens according to these rules

operand one	the other operand	common, new type
long double	anything	long double
double	anything	double
float	anything	float
unsigned long	anything	unsigned long
long	anything (int, unsigned)	long
unsigned	anything (int)	unsigned
int	anything (int)	int

Type conversions

Example for conversion

```
1 int a = 3;  
2 double b = 2.4;  
3 a = a*b;
```

Type conversions

Example for conversion

```
1 int a = 3;  
2 double b = 2.4;  
3 a = a*b;
```

1 $3 \rightarrow 3.0$

Type conversions

Example for conversion

```
1 int a = 3;  
2 double b = 2.4;  
3 a = a*b;
```

1 $3 \rightarrow 3.0$

2 $3.0 * 2.4 \rightarrow 7.2$

Type conversions

Example for conversion

```
1 int a = 3;  
2 double b = 2.4;  
3 a = a*b;
```

1 $3 \rightarrow 3.0$

2 $3.0 * 2.4 \rightarrow 7.2$

3 $7.2 \rightarrow 7$

Chapter 3

Pointers

Fundamental Theorem of Software Engineering (FTSE)

*“We can solve any problem
by introducing an extra level of indirection.”*

Andrew Koenig

Where are the variables?

Let's write a program that lists the address and value of variables

```
1 int a = 2;  
2 double b = 8.0;  
3 printf("address of a: %p, its value: %d\n", &a, a);  
4 printf("address of b: %p, its value: %f\n", &b, b);
```

```
address of a: 0x7fffa3a4225c, its value: 2  
address of b: 0x7fffa3a42250, its value: 8.000000
```

¹more precisely left-values

Where are the variables?

Let's write a program that lists the address and value of variables

```
1 int a = 2;  
2 double b = 8.0;  
3 printf("address of a: %p, its value: %d\n", &a, a);  
4 printf("address of b: %p, its value: %f\n", &b, b);
```

```
address of a: 0x7fffa3a4225c, its value: 2  
address of b: 0x7fffa3a42250, its value: 8.000000
```

- address of variable: starting address of "memory block" containing the variable, expressed in bytes
- with the address-of operator we can create address of any variables¹ like this `&<reference>`

¹more precisely left-values

The pointer type

The pointer type is for storing memory addresses

Declaration of pointer

```
<pointed type> * <identifier>;
```

```
1 int*    p; /* p stores the address of one int data */
2 double* q; /* q stores the address of one double data */
3 char*   r; /* r stores the address of one char data */
```

The pointer type

The pointer type is for storing memory addresses

Declaration of pointer

```
<pointed type> * <identifier>;
```

```
1 int*    p; /* p stores the address of one int data */
2 double* q; /* q stores the address of one double data */
3 char*   r; /* r stores the address of one char data */
```

it is the same, even if arranged in a different way

```
1 int     *p; /* p stores the address of one int data */
2 double *q; /* q stores the address of one double data */
3 char    *r; /* r stores the address of one char data */
```

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"

Operator of indirection

- If pointer p stores the address of variable a , then p "points to a "
- If p points to a , then variable a can be accessed as $*p$.
Here $*$ is the operator of indirection (dereference operator).

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```

a: ?? 0x1000

b: ?? 0x1004

p: ????

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```

a: ?? 0x1000

b: ?? 0x1004

p: ????

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```

a:	2	0x1000
----	---	--------

b:	??	0x1004
----	----	--------

p:	????
----	------

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```

a:	2	0x1000
----	---	--------

b:	??	0x1004
----	----	--------

p:	????
----	------

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```

a:	2	0x1000
----	---	--------

b:	3	0x1004
----	---	--------

p:	????
----	------

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```

a:	2	0x1000
----	---	--------

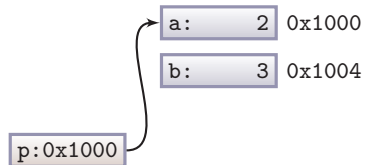
b:	3	0x1004
----	---	--------

p:	????
----	------

Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

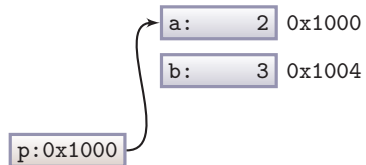
```
1  int a, b;  
2  int *p; /* int pointer */  
3  
4  a = 2;  
5  b = 3;  
6  p = &a; /* p points to a */  
7  *p = 4; /* a = 4 */  
8  p = &b; /* p points to b */  
9  *p = 5; /* b = 5 */
```



Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

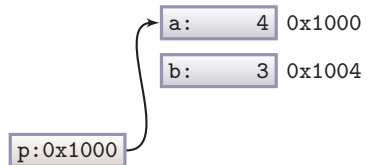
```
1  int a, b;  
2  int *p; /* int pointer */  
3  
4  a = 2;  
5  b = 3;  
6  p = &a; /* p points to a */  
7  *p = 4; /* a = 4 */  
8  p = &b; /* p points to b */  
9  *p = 5; /* b = 5 */
```



Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

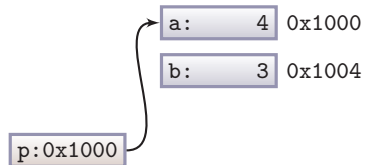
```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```



Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

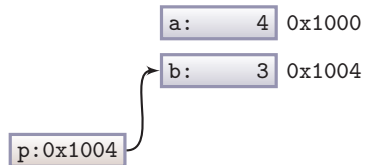
```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```



Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

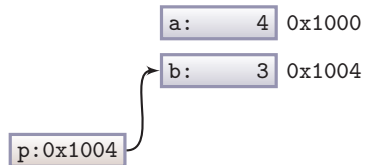
```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```



Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

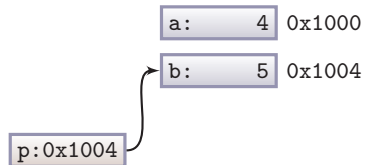
```
1  int a, b;  
2  int *p; /* int pointer */  
3  
4  a = 2;  
5  b = 3;  
6  p = &a; /* p points to a */  
7  *p = 4; /* a = 4 */  
8  p = &b; /* p points to b */  
9  *p = 5; /* b = 5 */
```



Operator of indirection

- If pointer `p` stores the address of variable `a`, then `p` "points to `a`"
- If `p` points to `a`, then variable `a` can be accessed as `*p`. Here `*` is the operator of indirection (dereference operator).

```
1 int a, b;  
2 int *p; /* int pointer */  
3  
4 a = 2;  
5 b = 3;  
6 p = &a; /* p points to a */  
7 *p = 4; /* a = 4 */  
8 p = &b; /* p points to b */  
9 *p = 5; /* b = 5 */
```



Address-of and indirection – summary

operator	operation	description
&	address-of	assigns its address to the variable
*	indirection	assigns variable to the address

Address-of and indirection – summary

operator	operation	description
&	address-of	assigns its address to the variable
*	indirection	assigns variable to the address

- Interpreting declaration: type of `*p` is `int`

```
1 int *p;      /* get used to this version */
```

Address-of and indirection – summary

operator	operation	description
<code>&</code>	address-of	assigns its address to the variable
<code>*</code>	indirection	assigns variable to the address

- Interpreting declaration: type of `*p` is `int`

```
1 int *p;      /* get used to this version */
```

- Multiple declaration: type of `a`, `*p` and `*q` is `int`

```
1 int a, *p, *q; /* at least because of this */
```

Application – Function for exchanging two variables

```
1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }
```

Application – Function for exchanging two variables

```
1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }
```

b 0x1FFC:

3

a 0x2000:

2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

0x1FF0:	15
0x1FF4:	2
0x1FF8:	3
b 0x1FFC:	3
a 0x2000:	2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

	0x1FF0:	15
x	0x1FF4:	2
y	0x1FF8:	3
b	0x1FFC:	3
a	0x2000:	2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

tmp 0x1FEC:	2
0x1FF0:	15
x 0x1FF4:	2
y 0x1FF8:	3
b 0x1FFC:	3
a 0x2000:	2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

tmp 0x1FEC:	2
0x1FF0:	15
x 0x1FF4:	3
y 0x1FF8:	3
b 0x1FFC:	3
a 0x2000:	2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

tmp 0x1FEC:	2
0x1FF0:	15
x 0x1FF4:	3
y 0x1FF8:	2
b 0x1FFC:	3
a 0x2000:	2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

	0x1FF0:	15
x	0x1FF4:	3
y	0x1FF8:	2
b	0x1FFC:	3
a	0x2000:	2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

0x1FF0:	15
0x1FF4:	3
0x1FF8:	2
b 0x1FFC:	3
a 0x2000:	2

Application – Function for exchanging two variables

```
1 void xchg(int x, int y) {  
2     int tmp = x;  
3     x = y;  
4     y = tmp;  
5 }  
6  
7 void xchgp(int *px, int *py) {  
8     int tmp = *px;  
9     *px = *py;  
10    *py = tmp;  
11 }  
12  
13 int main(void) {  
14     int a = 2, b = 3;  
15     xchg(a, b);  
16     /* NO exchange */  
17     xchgp(&a, &b); /* exchange */  
18     return 0;  
19 }
```

b 0x1FFC:	3
a 0x2000:	2

Application – Function for exchanging two variables

```
1 void xchg(int x, int y) {  
2     int tmp = x;  
3     x = y;  
4     y = tmp;  
5 }  
6  
7 void xchgp(int *px, int *py) {  
8     int tmp = *px;  
9     *px = *py;  
10    *py = tmp;  
11 }  
12  
13 int main(void) {  
14     int a = 2, b = 3;  
15     xchg(a, b);  
16     /* NO exchange */  
17     xchgp(&a, &b); /* exchange */  
18     return 0;  
19 }
```

b 0x1FFC:

3

a 0x2000:

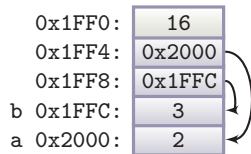
2

Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

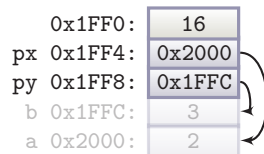


Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

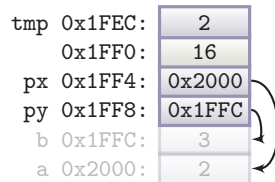


Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

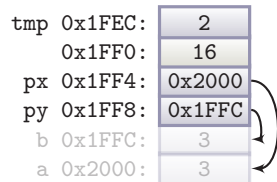


Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

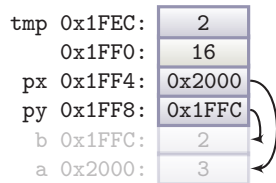


Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

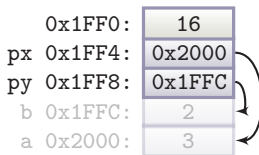


Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```

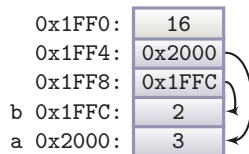


Application – Function for exchanging two variables

```

1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }

```



Application – Function for exchanging two variables

```
1 void xchg(int x, int y) {  
2     int tmp = x;  
3     x = y;  
4     y = tmp;  
5 }  
6  
7 void xchgp(int *px, int *py) {  
8     int tmp = *px;  
9     *px = *py;  
10    *py = tmp;  
11 }  
12  
13 int main(void) {  
14     int a = 2, b = 3;  
15     xchg(a, b);  
16     /* NO exchange */  
17     xchgp(&a, &b); /* exchange */  
18     return 0;  
19 }
```

b 0x1FFC:	2
a 0x2000:	3

Application – Function for exchanging two variables

```
1 void xchg(int x, int y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
6
7 void xchgp(int *px, int *py) {
8     int tmp = *px;
9     *px = *py;
10    *py = tmp;
11 }
12
13 int main(void) {
14     int a = 2, b = 3;
15     xchg(a, b);
16     /* NO exchange */
17     xchgp(&a, &b); /* exchange */
18     return 0;
19 }
```

Application – returning value as parameter

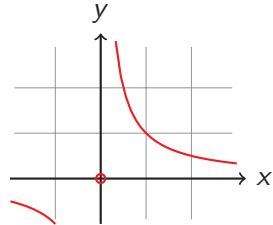
- If a function has to calculate several values, then...
...we can use structures, but sometimes this seems rather unnecessary.

Application – returning value as parameter

- If a function has to calculate several values, then...
 - ...we can use structures, but sometimes this seems rather unnecessary.
 - Instead...

```
1 int inverse(double x, double *py)
2 {
3     if (abs(x) < 1e-10) return 0;
4     *py = 1.0 / x;
5     return 1;
6 }
```

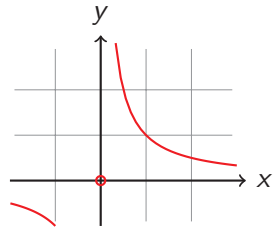
[link](#)



Application – returning value as parameter

- If a function has to calculate several values, then...
 - ...we can use structures, but sometimes this seems rather unnecessary.
 - Instead...

```
1 int inverse(double x, double *py)
2 {
3     if (abs(x) < 1e-10) return 0;
4     *py = 1.0 / x;
5     return 1;
6 }
```

[link](#)

```
1 double y;          /* memory allocation for result */
2 if (inverse(5.0, &y) == 1)
3     printf("Reciprocal of %f is %f\n", 5.0, y);
4 else
5     printf("Reciprocal does not exist");
```

[link](#)

Application – return values as parameters

- Now we understand what this means

```
1 int n, p;  
2 /* return value as parameter */  
3 scanf("%d%d", &n, &p); /* we pass the addresses */
```

Remarks:

- What is the use of having different pointer types for different types?

Remarks:

- What is the use of having different pointer types for different types?
- Type = set of values + operations
- Obviously set of values is the same for all pointers (unsigned integer addresses)
- Operations are different!

Remarks:

- What is the use of having different pointer types for different types?
- Type = set of values + operations
- Obviously set of values is the same for all pointers (unsigned integer addresses)
- Operations are different!
- The operator of indirection (*)
 - makes `int` from `int` pointer
 - makes `char` from `char` pointer

Remarks:

- What is the use of having different pointer types for different types?
- Type = set of values + operations
- Obviously set of values is the same for all pointers (unsigned integer addresses)
- Operations are different!
- The operator of indirection (*)
 - makes `int` from `int` pointer
 - makes `char` from `char` pointer
- Other differences are detailed in pointer-arithmetics. . .

Pointer-arithmetics

If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

Pointer-arithmetics

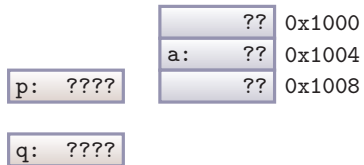
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

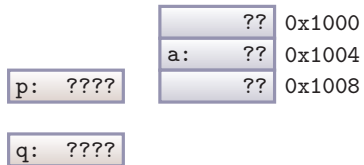
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

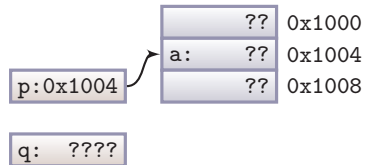
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

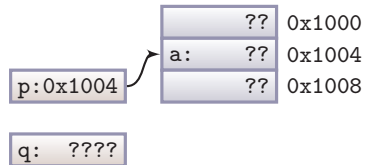
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

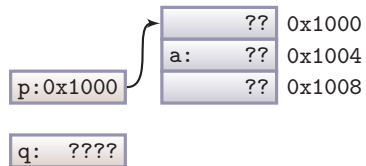
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

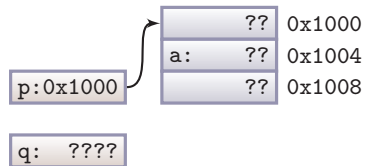
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

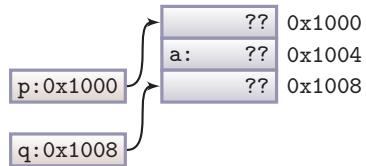
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

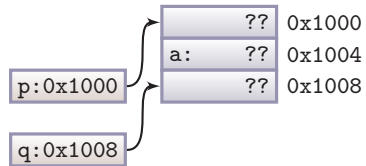
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



Pointer-arithmetics

If p and q are pointers of the same type, then

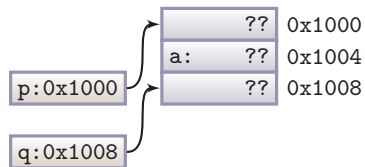
expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```

2



Pointer-arithmetics

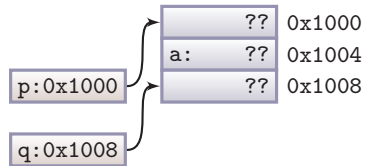
If p and q are pointers of the same type, then

expr.	type	meaning
$p+1$	pointer	points to the next <u>element</u>
$p-1$	pointer	points to the previous <u>element</u>
$q-p$	integer number	number of <u>elements</u> between two addresses

```

1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);

```



2

At pointer-arithmetic operations addresses are "measured" in

Pointer-arithmetic

- In the above example pointer-arithmetic is strange, as we don't know what is before or after variable `a` in the memory.
- This operation is meaningful, when we have variables of the same type, stored in the memory one after the other.
- This is the case for arrays.

Pointers and arrays

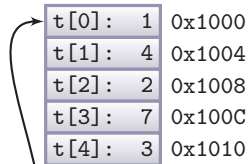
- Traversing an array can be done with pointer-arithmetics.

Pointers and arrays

- Traversing an array can be done with pointer-arithmetics.

```
1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = &t[0];
5 for (i = 0; i < 5; ++i)
6     printf("%d ", *(p+i));
```

1 4 2 7 3



t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010


p:0x1000

Pointers and arrays

- Traversing an array can be done with pointer-arithmetics.

```
1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = &t[0];
5 for (i = 0; i < 5; ++i)
6     printf("%d ", *(p+i));
```

1 4 2 7 3



t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000

- In this example $*(p+i)$ is the same as $t[i]$, because p points to the beginning of array t

Pointers and arrays

- Pointers can be taken as arrays, this means they can be indexed.
By definition $p[i]$ is identical to $*(p+i)$

Pointers and arrays

- Pointers can be taken as arrays, this means they can be indexed.


By definition $p[i]$ is identical to $*(p+i)$

```
1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = &t[0];
5 for (i = 0; i < 5; ++i)
6     printf("%d ", p[i]);
```

1 4 2 7 3

t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000



Pointers and arrays

- Pointers can be taken as arrays, this means they can be indexed.


By definition $p[i]$ is identical to $*(p+i)$

```
1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = &t[0];
5 for (i = 0; i < 5; ++i)
6     printf("%d ", p[i]);
```

1 4 2 7 3

t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000



- In this example $p[i]$ is the same as $t[i]$, because p points to the beginning of array t

Pointers and arrays

- Arrays can be taken as pointers.
The identifier (name) of array is the starting address of the array, in other words the value of expression `t` is `&t[0]`

Pointers and arrays

- Arrays can be taken as pointers.

The identifier (name) of array is the starting address of the array, in other words the value of expression `t` is `&t[0]`

```
1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = t; /* &t[0] */
5 for (i = 0; i < 5; ++i)
6     printf("%d ", p[i]);
```

1 4 2 7 3

t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000


Pointers and arrays

- Arrays can be taken as pointers.

The identifier (name) of array is the starting address of the array, in other words the value of expression `t` is `&t[0]`

```
1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = t; /* &t[0] */
5 for (i = 0; i < 5; ++i)
6     printf("%d ", p[i]);
```

1 4 2 7 3



t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000

- Pointer-arithmetics work for arrays too:
`t+i` is identical to `&t[i]`

Pointers and arrays – summary

- Pointer can be taken as array, and array as a pointer.

Pointers and arrays – summary

- Pointer can be taken as array, and array as a pointer.
- index operator is only a notation
the compiler will **always** replace $a[i]$ with $*(a+i)$,
both if a is pointer, and also if a is array.

Pointers and arrays – summary

- Pointer can be taken as array, and array as a pointer.
- index operator is only a notation
the compiler will **always** replace $a[i]$ with $*(a+i)$,
both if a is pointer, and also if a is array.
- Differences:
 - Elements of array have allocated space in memory (variables).
No allocated elements belong to the pointer.

Pointers and arrays – summary

- Pointer can be taken as array, and array as a pointer.
- index operator is only a notation
the compiler will **always** replace `a[i]` with `*(a+i)`,
both if `a` is pointer, and also if `a` is array.
- Differences:
 - Elements of array have allocated space in memory (variables).
No allocated elements belong to the pointer.
 - Starting address of array is constant, it cannot be changed.
Pointer is a variable, the address stored in it can be modified.

Pointers and arrays – summary

- Pointer can be taken as array, and array as a pointer.
- index operator is only a notation
the compiler will **always** replace `a[i]` with `*(a+i)`,
both if `a` is pointer, and also if `a` is array.
- Differences:
 - Elements of array have allocated space in memory (variables).
No allocated elements belong to the pointer.
 - Starting address of array is constant, it cannot be changed.
Pointer is a variable, the address stored in it can be modified.

```

1  int array[5] = {1, 3, 2, 4, 7};
2  int *p = array;
3
4  /* the elements can be accessed via p and a */
5  p[0] = 2;           array[0] = 2;
6  *p = 2;            *array = 2;
7
8  /* p can be changed   array CANNOT */
9  p = p+1; /* ok */     array = array + 1; /* ERROR */

```

Passing arrays to functions

- Let's use a function to determine the first negative element of array!

³defined in `stdio.h`

Passing arrays to functions

- Let's use a function to determine the first negative element of array!
- Passing an array:
 - Address of first element `double*`
 - Size of the array `typedef unsigned int size_t`³

³defined in `stdio.h`

Passing arrays to functions

- Let's use a function to determine the first negative element of array!
- Passing an array:
 - Address of first element `double*`
 - Size of the array `typedef unsigned int size_t3`

```
1 double first_negative(double *array, size_t size)
2 {
3     size_t i;
4     for (i = 0; i < size; ++i) /* for each elems. */
5         if (array[i] < 0.0)
6             return array[i];
7
8     return 0; /* all are non-negative */
9 }
```

[link](#)

```
1 double myarray[3] = {3.0, 1.0, -2.0};
2 double neg = first_negative(myarray, 3);
```

[link](#)

³defined in `stdio.h`

Passing arrays to functions

- To distinguish arrays and pointers in the parameter list, we can use the array-notation when passing an array.

```
1 double first_negative(double array[], size_t size)
2     /* (double *array, size_t size) */
3 {
4     ...
5 }
```

Passing arrays to functions

- To distinguish arrays and pointers in the parameter list, we can use the array-notation when passing an array.

```
1 double first_negative(double array[], size_t size)
2     /* (double *array, size_t size) */
3 {
4     ...
5 }
```

- In the formal parameter list `double a[]` is identical to `double *a`.
- In the formal parameter list we can use only empty `[]`, and size should be passed as a separate parameter!

Passing arrays to functions

- Let's use a function to determine the first negative element of array!
- The return value should be the **address** of the element found.

```
1 double *first_negative(double *array, size_t size)
2 {
3     size_t i;
4     for (i = 0; i < size; ++i) /* for each elems. */
5         if (array[i] < 0.0)
6             return &array[i];
7
8     return NULL; /* all are non-negative */
9 }
```

[link](#)

Passing arrays to functions

- Let's use a function to determine the first negative element of array!
- The return value should be the **address** of the element found.

```
1 double *first_negative(double *array, size_t size)
2 {
3     size_t i;
4     for (i = 0; i < size; ++i) /* for each elems. */
5         if (array[i] < 0.0)
6             return &array[i];
7
8     return NULL; /* all are non-negative */
9 }
```

[link](#)

Null pointer

- The null pointer (NULL)

Null pointer

- The null pointer (NULL)
 - It stores the 0x0000 address

Null pointer

- The null pointer (NULL)
 - It stores the 0x0000 address
 - Agreed that it "points to nowhere"

Thank you for your attention.