Basics of C

Lecture 5, final

Summer 2018
Innopolis University
Eugene Zouev

Previous classes:

- C memory model
- Typical program structure
- Declarations & types
- Pointers, arrays
- Global/local & dynamic objects
- Static/auto & global/local objects
- Structures & unions
- Memory model: execution stack
- Scopes & blocks

Today:

- Statements
- Expressions
- Preprocessor

Statements in C

The "standard" set of statements

- Selection statements: if & switch
- Iteration statements: for-, while- & do-loops
- Jump statements: goto, return, break & continue
- Compound statements, or blocks
- Empty, or null statements

What's missing? ☺

- ~~Assignments~~
- ~~Function calls~~

Statements in C

In addition to the “standard” set there are two non-conventional kinds of statements:

- **Declaration statement**
- **Expression statement**

To be more precise:

Compound statement is a **sequence of statements and/or declarations**; therefore, a declaration within a block is considered **as a statement**.

What is “expression statement”:

```
expression ;
```

The common rule: Statements do not issue values (do not produce results)

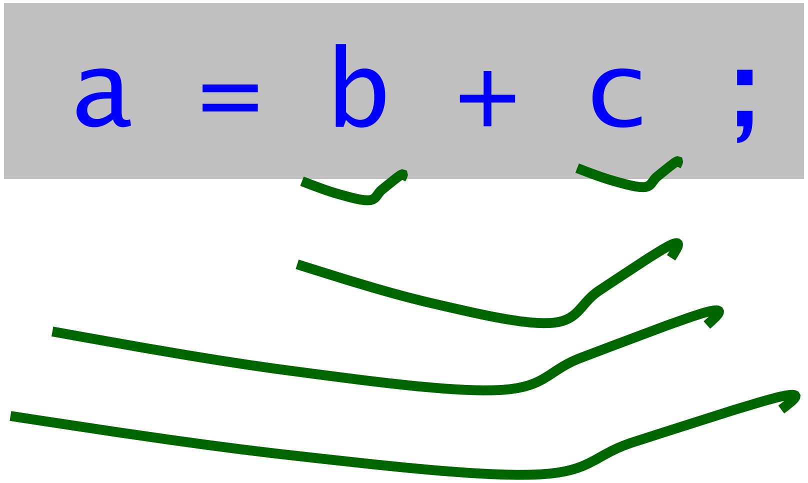
Statements in C

```
int f()
{
   int i = 3;
   for ( int j=0; j<20; j++ )
   {
      int k = 0;
      if ( j<=10 )
      {
        int i = 7;
        k += g(i); *
      }
      else
      {
        h(k+i); **
        int j = g(k+i);  ***
        k += j*i;
      }
   }
   return k*k;
}
```

Comments

- Function body is a block (compound statement). It contains a sequence of statements. Both "ordinary" and declaration statements are in the block.
- The body of the for-loop is (also) a block with one declaration statement and one "ordinary" statement.
- **(*)** This is the expression statement. It contains the expression. The value of the expression **is discarded** (only **side effect** of the expression statement matters).
- **(**)** This is also the expression statement. It contains the function call. If the h function returns the value, **it's discarded**.
- **(***)** This is declaration statement. Here, the position of the declaration is not at the beginning of the block – being a statement, it can occur at any position within the block.

Expression Statements



```
a = b + c ;
```

Therefore, the assignment is used only for its side effect.

```
f();
```

The same: even if f returns a value, it is discared.

Semantics

- b and c are expressions. The results of their execution is the current values of corresponding variables.
- b+c is the expression. The result of the expression is the sum of results of calculations of b and c.
- a is the expression. The result of the expression is **reference to memory** where the current value of a is stored.
- a=b+c is the expression called **assignment**. It's semantics is as follows: the value of the right-part expression gets assigned by the reference to memory denoted by the left-part expression. **The result of the assignment is the value of its right-part expression**.
- a=b+c; is the **expression statement**. Its semantics is: the containing expression (assignment) is executed, and its result (if any) **is discarded**.

Expressions in C

“Expression” is a formula for calculating values.

- Any expression (almost any ☺) issues a value.

In general, expressions are built of

- Operands
- Operators
- Parentheses

using ordinary rules (as in many other programming languages).

```
f()*(a+b)-*p++;
```

Expressions in C

Primary expression elements:

- Identifier (designates a variable/constant/function)

```
fun   abs   ptr_fun
```

Identifiers designate corresponding entities:
Either values of variables/constants or function addresses

- Literal: integer/floating/string

```
123   0xFE   0.01E-2   "string"
```

Literals designate themselves

- Subexpression enclosed in parentheses

```
(a-b)
```

Subexpressions designate values of enclosed expressions.

Expressions in C

Secondary expression elements ("postfix expressions") - are built on top of primary expressions:

- Array subscripting

```
arr[i+j*2]
```

Value of or reference to an array element.

- Function call

```
fun(*p,&x,777+y)
```

The result of the function call.

- Structure/union member access

```
ptr->m    s.m
```

Value of or reference to a struct member.

- Postfix decrement/increment

```
ptr--    arr++
```

The result is the initial pointer (**YES**!)
The side effect: the pointer gets moved to the previous/next element depending of the type pointer to by the pointer

Expressions in C

Next (higher-level) building blocks: <u>unary expressions</u>
- are built on top of postfix expressions:

- Prefix increment/decrement

```
p--     ++x
```

Result: the value of the operand increased or decreased by one.

- Address & indirection

```
&x     *(p+1)
```

Result: the address of the operand OR the value pointed to by the pointer from the operand.

- Unary plus/minus

```
+x     -v
```

Value of or reference to a struct member.

- Bitwise complement & logical negation

```
~v     !v
```

The result: the initial value inverted or negated

- Sizeof operator

```
sizeof (T)   sizeof a+b
```

The result: integer value

Expressions in C

The highest-level building blocks for expressions:
binary expressions:

- Additive & multiplicative operators

```
a+b     b-c     c*d     d/e     e%f
```

- Relational & equality operators

```
a<b     a<=b     a>b     a>=b     a==b     a!=b
```

- Bitwise shift operators

```
a << b        a >> b
```

- Bitwise logical operators

```
a & b       a | b       a ^ b
```

- Logical operators

```
a && b       a || b
```

Expressions in C

These are also binary operators:

- Assignment operators

```
a = b
```

```
a+=b   a-=b   a*=b   a/=b   a%=b
<<=   >>=
&=   ^=   |=
```

- **Comma operator (!!)**

```
expr1 , expr2
```

The left expression is evaluated; its value **is discarded**. Then the second expression is evaluated. Its value is the result of the whole comma expression.

- Conditional operator

```
expression ? expression : expression
```

The single **ternary** operator in the language

Expressions in C

Basic rules for expressions

- Unary operators are performed **from right to left**.

```
&*p    ~-v    *f()
```

- Binary operators are performed **in accordance with their preferences**.

```
a[i] + b * *p
```

- Binary operators of the same preference are performed **from left to right**.

```
x + y - z
```

```
a[i] = b = c + d*e
```

- The **side effect** of the expression (if any) happens after both operands are evaluated.

```
a[i++] = i
```

- Parentheses are used to change the default execution order.

```
(a[i] + b) * *p
```

Expressions in C

Some examples for the comma operator

```
if ( f(b),g(c) )
   ...
else
   ...
```

```
for ( int i=0, j=0; i<10 && j<10; i++, j++)
{
  Some calculations on a matrix…
}
```

Expressions in C

Some more examples ☺.

- Suppose p1 and p2 are pointers.

```
while (*p1++ = *p2++);
```

The loop performs **copying** elements of one array/string to another until the element of value 0 is encountered.

```
while (*p1++ == *p2++);
```

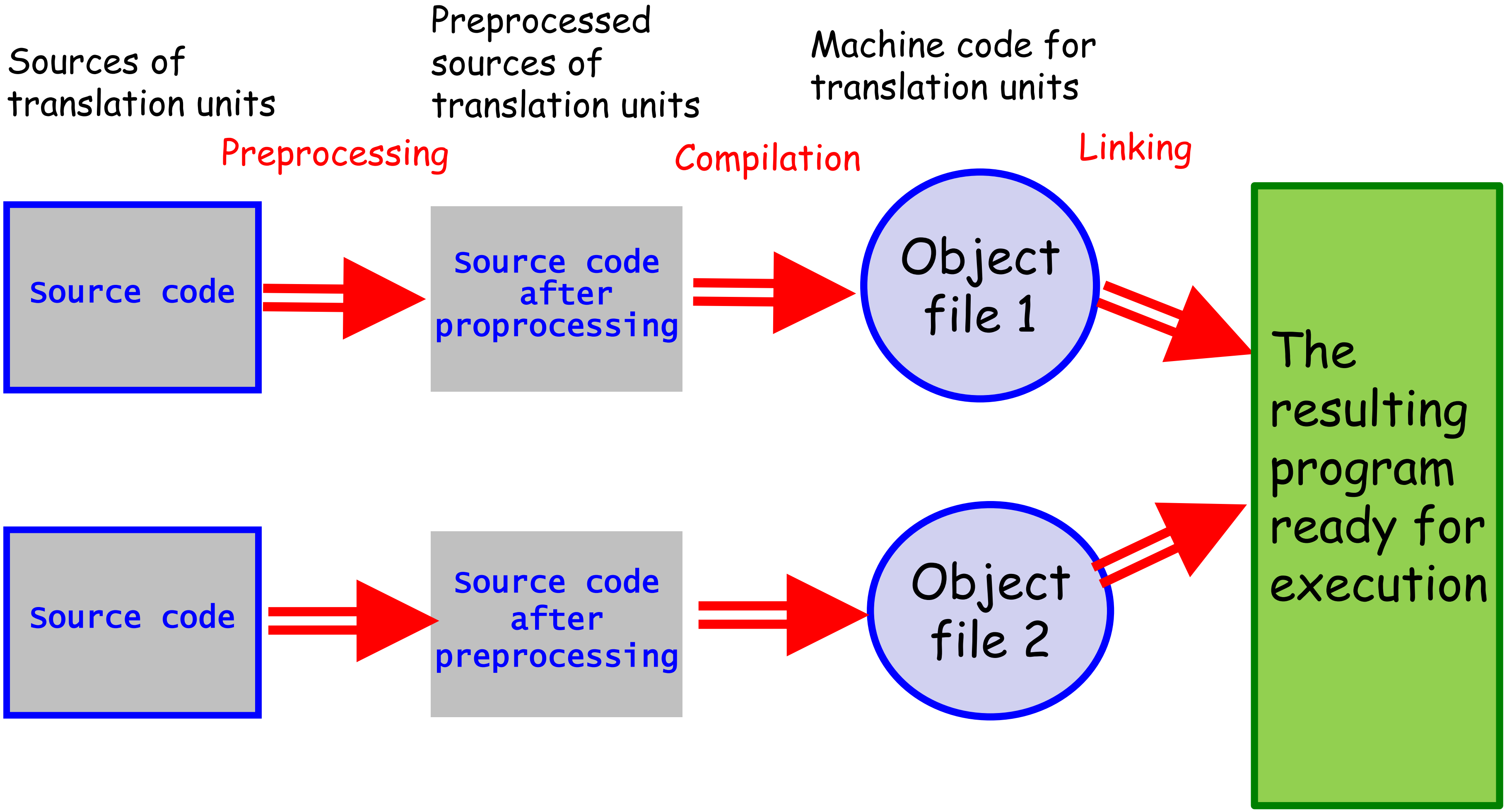
The loop performs **comparison** elements of one array/string with corresponding elements of another. The loop stops when the first pair of non-equal elements is encountered.

- This is the while loop.
- The construct within parentheses is the expression that specifies the loop condition: whether to continue executing loop body or to exit the loop.
- The body of the loop contains only one statement: this is empty statement. It doesn't perform any actions.
- Therefore, all useful actions are within the loop header.

Preprocessing

How C Programs are Built

The full picture



Sources of translation units

Preprocessed sources of translation units

Machine code for translation units

Preprocessing

Compilation

Linking

Source code

Source code after proprocessing

Object file 1

Source code

Source code after preprocessing

Object file 2

The resulting program ready for execution

Preprocessing

"I would have killed preprocessor"
- B.Stroustrup

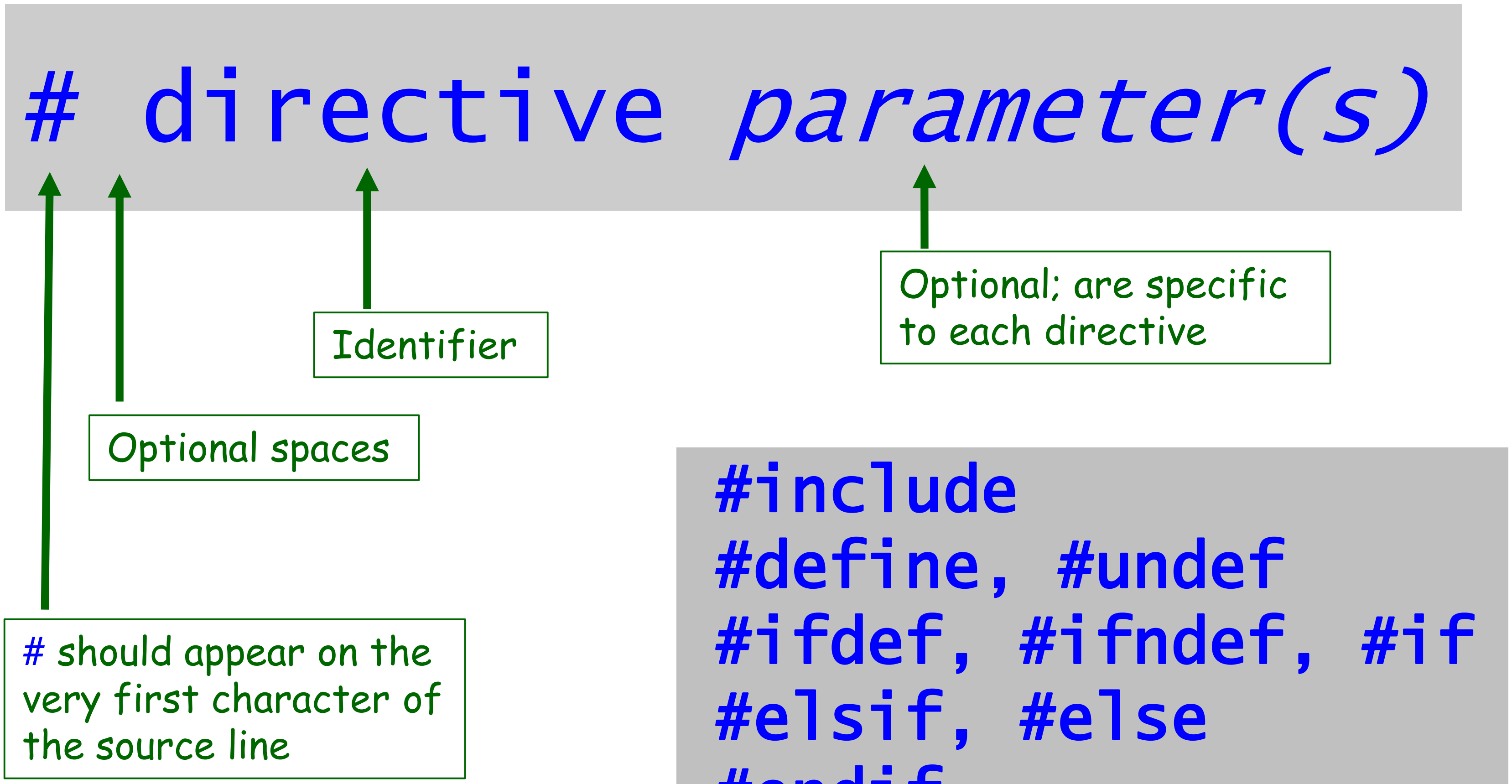
Major ideas and constructs

- This is purely **text-to-text** processing.
- **No C syntax/semantic rules** are involved.
- Preprocessing is usually performed by a separate tool – **preprocessor** – following its own rules.
- The main preprocessing mechanism is **text substitution**: some parts of the source text get replaced for other texts. Substitution process is usually **repetitive**.
- Main constructs:
  * **preprocessing directives**
  * **macro calls**
- The most popular preprocessing directive:
  `#include "filename"`

Preprocessing directives

They specify rules for substitution

Common syntax



```
# directive parameter(s)
```

Identifier

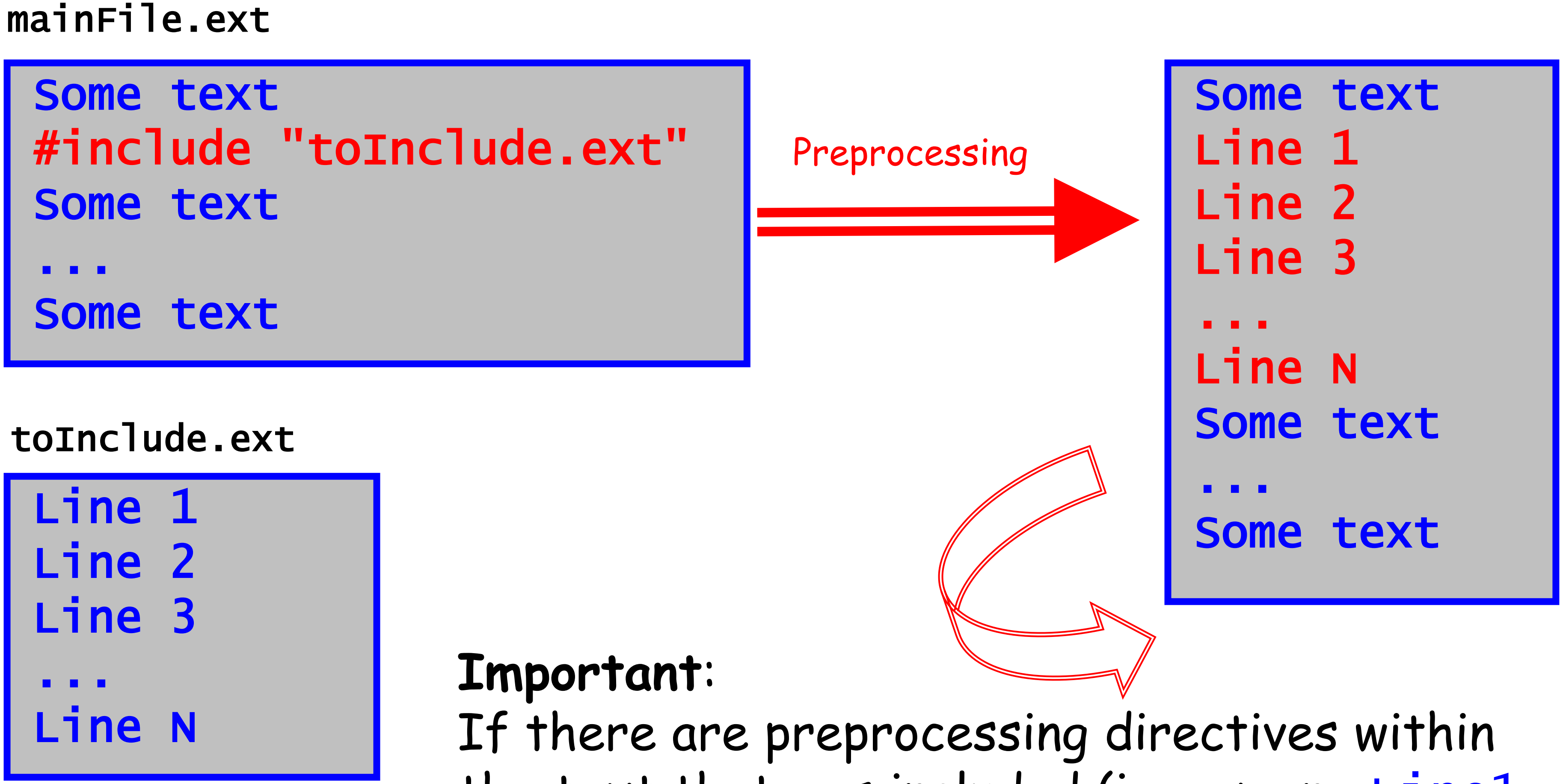
Optional; are specific to each directive

Optional spaces

# should appear on the very first character of the source line

```
#include
#define, #undef
#ifdef, #ifndef, #if
#elsif, #else
#endif
...
```

#include: Text inclusion



mainFile.ext

```
Some text
#include "toInclude.ext"
Some text
...
Some text
```

toInclude.ext

```
Line 1
Line 2
Line 3
...
Line N
```

```
Some text
Line 1
Line 2
Line 3
...
Line N
Some text
...
Some text
```

Important:
If there are preprocessing directives within the text that was included (i.e., among `Line1`, …, `LineN`) then the resulting text **in turn gets preprocessed**.

#define: Macro definition

Syntax

Macro name

Macro body

```
#define MacroId sequence-of-tokens
```

```
#define MacroId(ParId-1,…,ParId-N) sequence-of-tokens
```

Semantics: macro expansion

1. For each occurrence of `MacroId` in the source text, it gets replaced for the sequence of tokens specified after it.

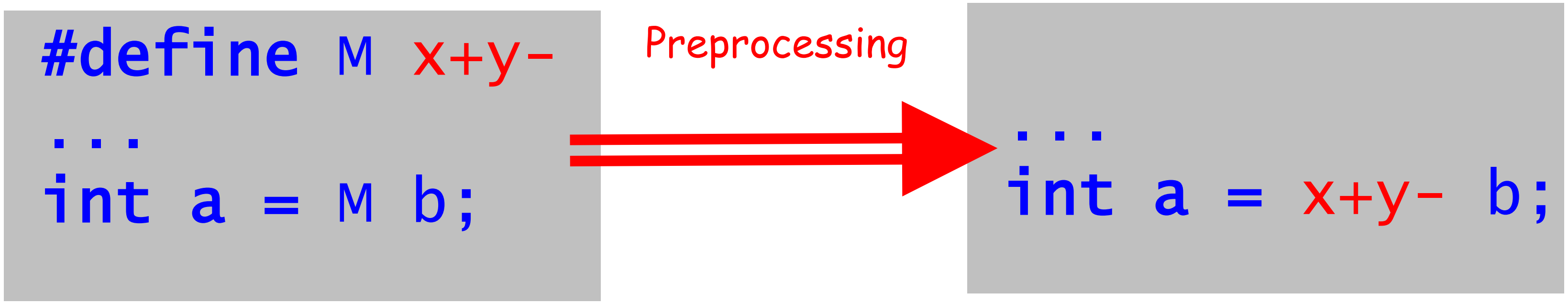
2. For each occurrence of the construct like

   ```
   MacroId(SeqOfTokens-1,…,SeqOfTokens-N)
   ```

   it gets replaced for the sequence of tokens, **and** each occurrence of `ParId-i` in the macro body gets replaced for the corresponding token sequence `SeqOfTokens-i`.

#define: Macro expansion

Examples

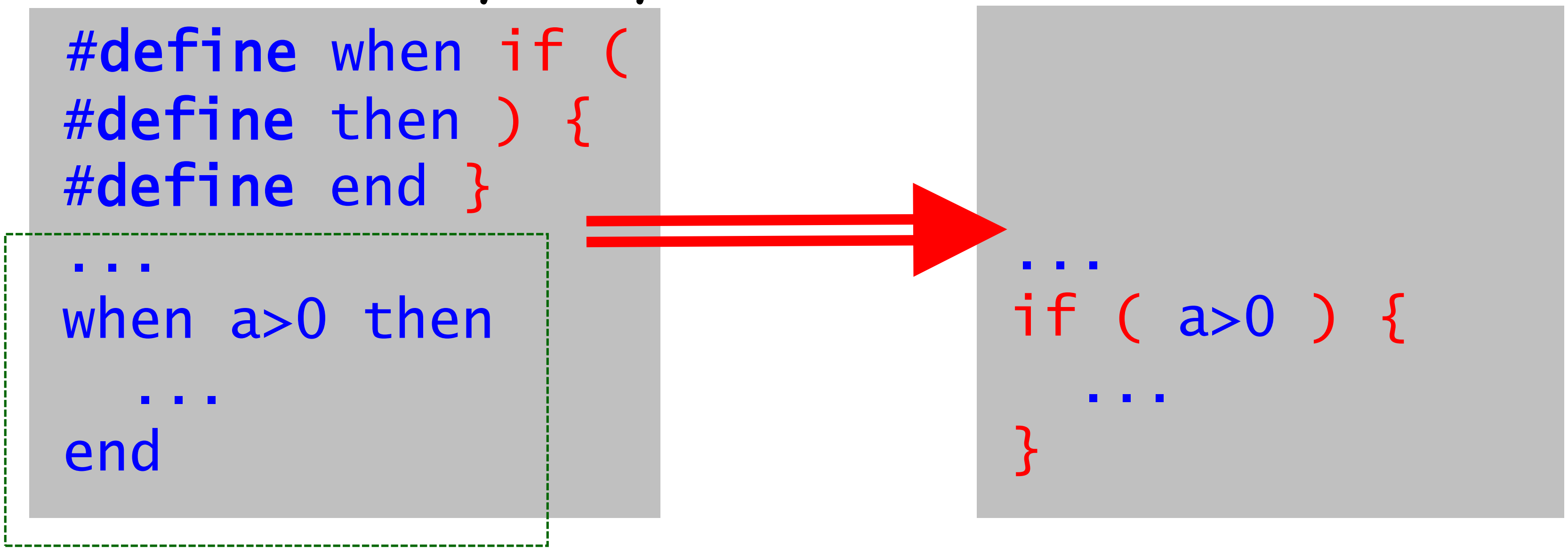


```
#define M x+y-
...
int a = M b;
```

Preprocessing

```
...
int a = x+y- b;
```

How to modify *C* syntax ☺



```
#define when if (
#define then ) {
#define end }
...
when a>0 then
  ...
end
```

```
...
if ( a>0 ) {
  ...
}
```

#define: Macro expansion

Examples

```
#define Max(a,b) a > b ? a : b
...
int test = Max(x,7);
int res1 = Max(x+y,x-y);
int res2 = Max(x+=y,x*=y);
```

Preprocessing

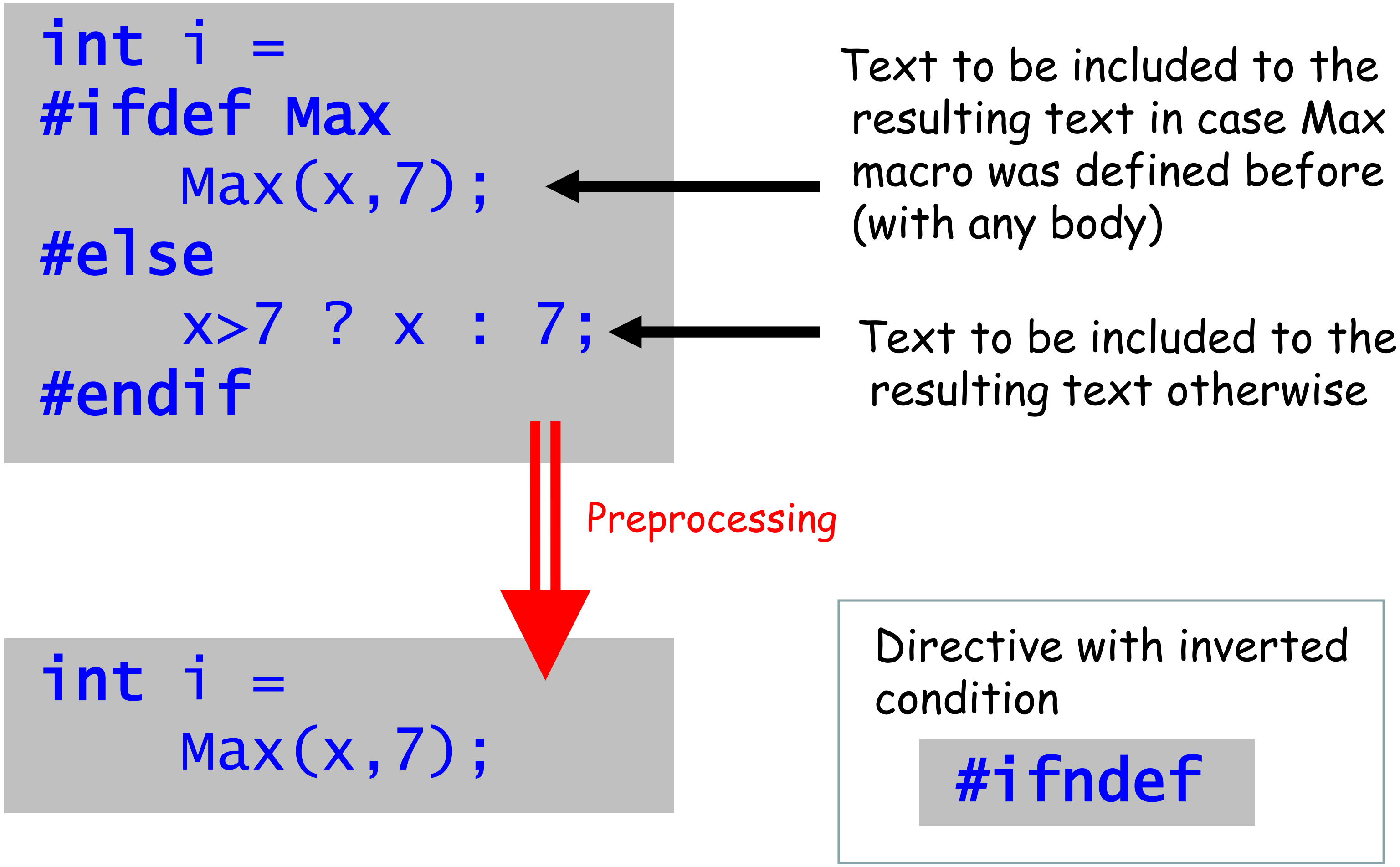
```
int test = x > 7 ? x : 7;

int res1 = x+y > x-y ? x+y : x-y;

int res2 = x+=y > x*=y ? x+=y : x*=y;
```

Is it the valid result??

Solution

```
#define Max(a,b) (a) > (b) ? (a) : (b)
```

#if(n)def: Conditional inclusion



```
int i =
#ifdef Max
    Max(x,7);
#else
    x>7 ? x : 7;
#endif
```

Text to be included to the resulting text in case Max macro was defined before (with any body)

Text to be included to the resulting text otherwise

Preprocessing

```
int i =
    Max(x,7);
```

Directive with inverted condition

```
#ifndef
```

#ifdef & #undef

Syntax

```
#undef MacroId
```

Existing macro name

Semantics

Just removes the macro `MacroId`

Example

```
#ifdef Max
#undef Max
int Max(int a, int b)
{
    return a>b ? a : b;
}
#endif
```

#include & #ifndef

mainFile.ext

```
Some text
#include "toInclude.ext"
#include "toInclude.ext"
Some text
...
Some text
```

How to prevent duplication??

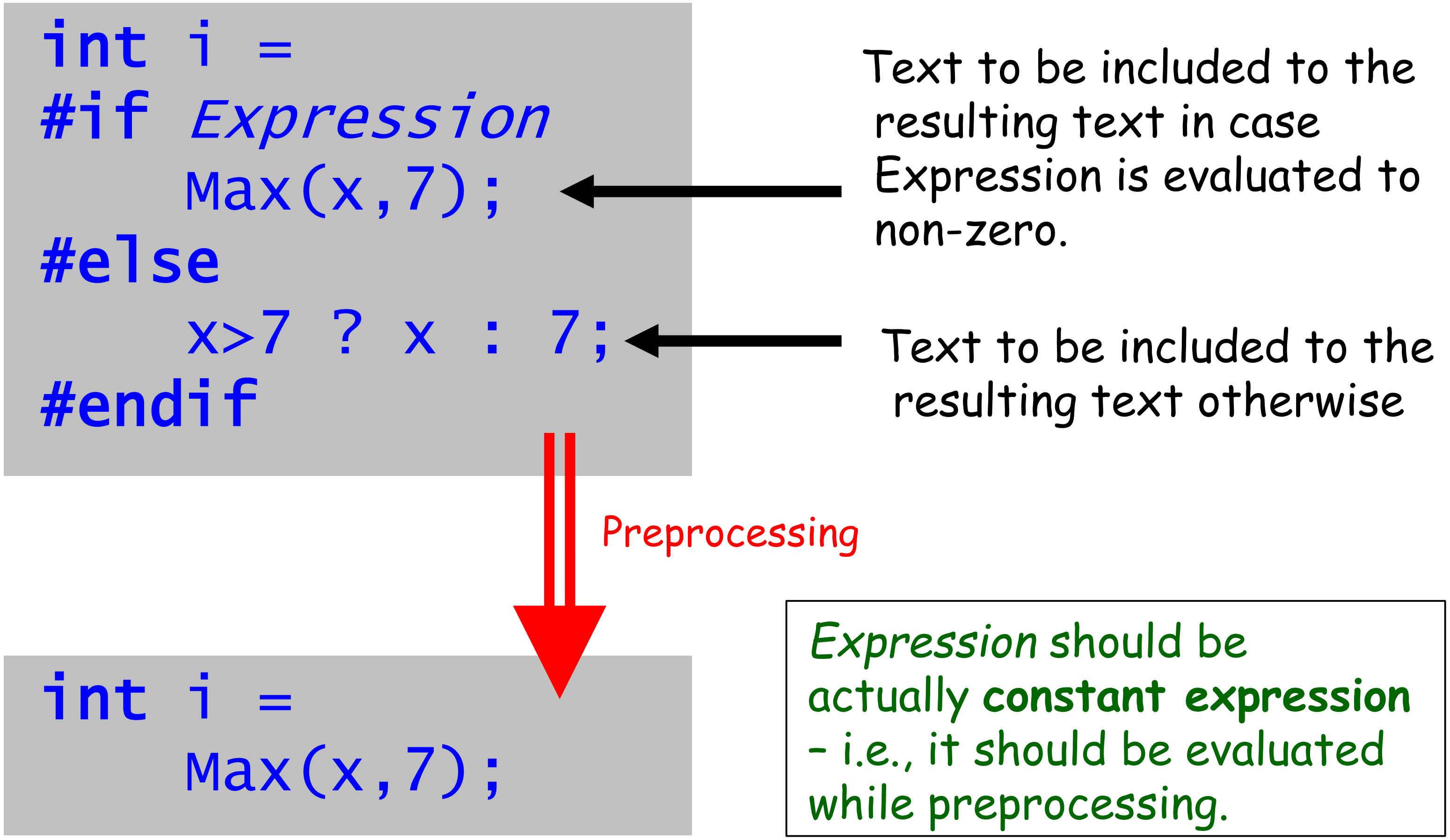
toInclude.ext

```
#ifndef __toInclude
#define __toInclude
Line 1
Line 2
Line 3
...
Line N
#endif
```

The solution

#if: Conditional inclusion

Semantics of #if is the same as for #ifdef but the value of **an expression** is checked



```
int i =
#if Expression
      Max(x,7);
#else
      x>7 ? x : 7;
#endif
```

Text to be included to the resulting text in case Expression is evaluated to non-zero.

Text to be included to the resulting text otherwise

Preprocessing

```
int i =
      Max(x,7);
```

Expression should be actually **constant expression** – i.e., it should be evaluated while preprocessing.

Predefined macros

And an example of **#if**

Example

```
#define __STDC_ISO_10646__ 199901L
```

- Predefined in the ISO/IEC Standard 9899:1999

```
#if __STDC_ISO_10646__ >= 199901L
   Some code that's specific to the
   version of C of 1999 or later
#elsif __STDC_ISO_10646__ >= 199409L
   Equivalent code legal in the
   previous version of the C standard
#else
   Equivalent code for an older
   version of C
#endif
```