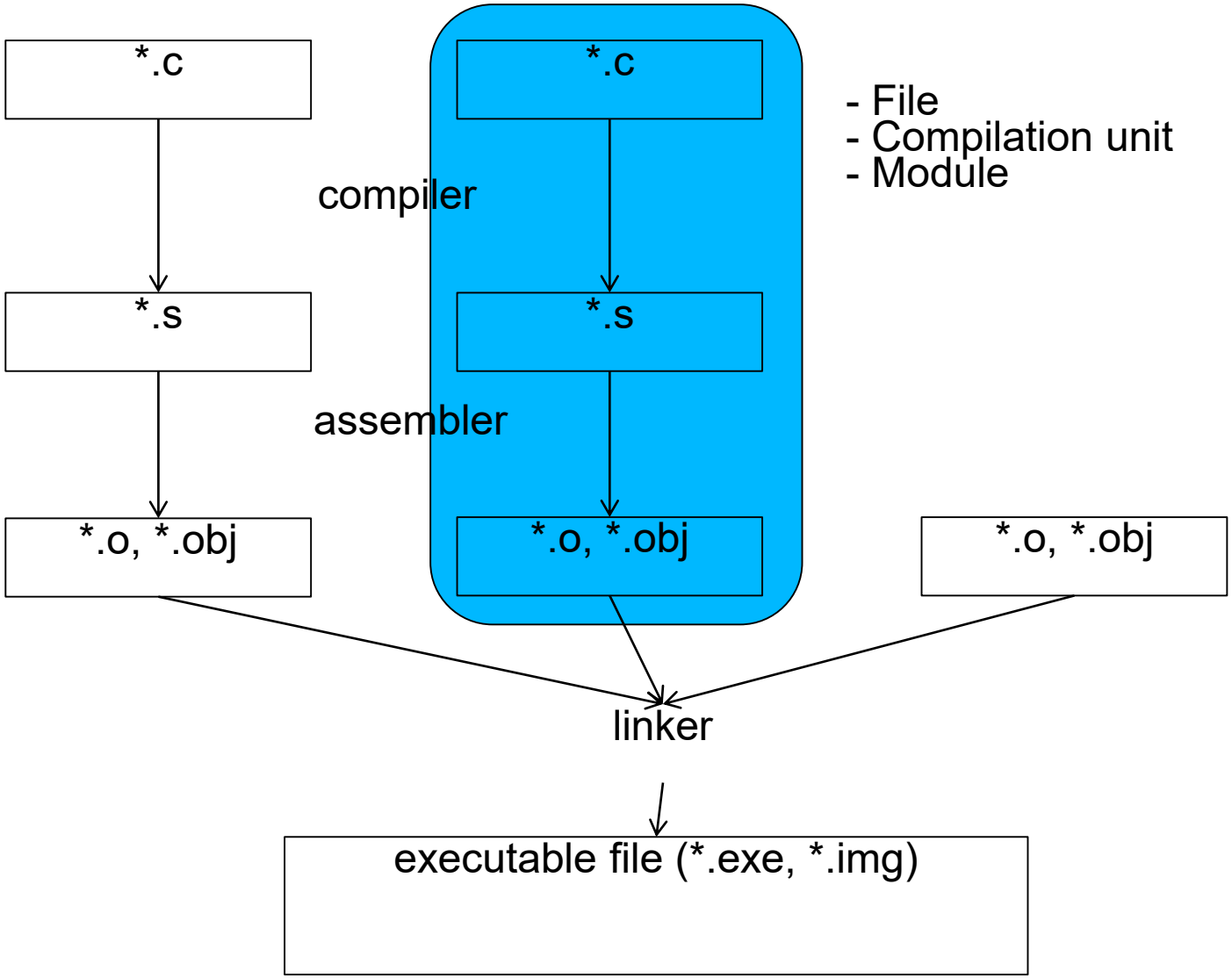
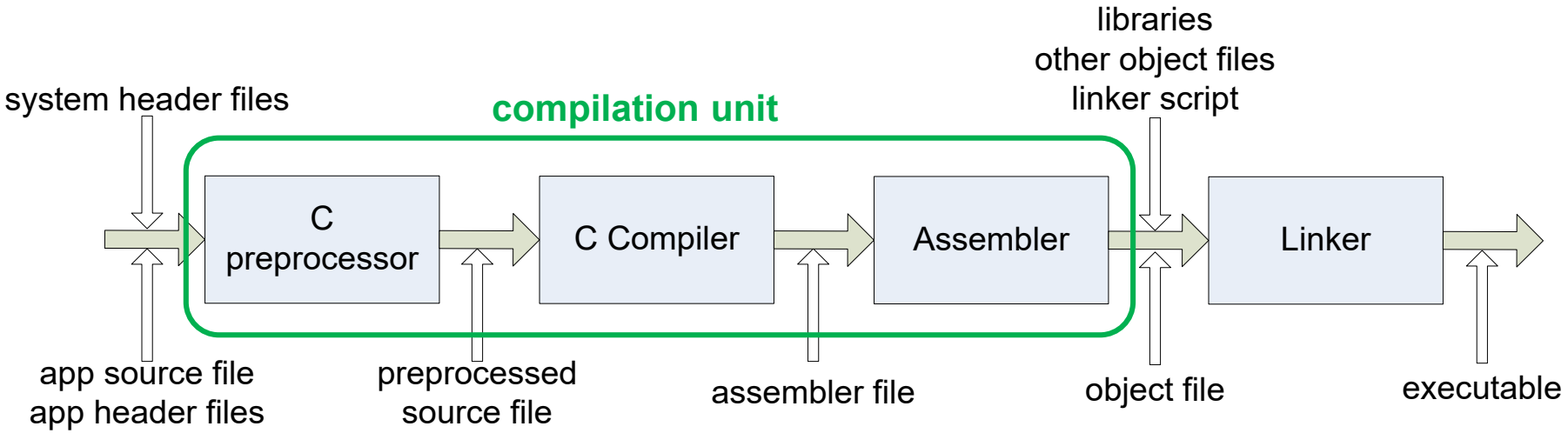
Build flow


*.c
*.s
*.o, *.obj
compiler
assembler
*.c
*.s
*.o, *.obj
- File
- Compilation unit
- Module
*.o, *.obj
linker
executable file (*.exe, *.img)

Build flow



name.c – source files

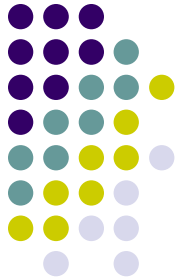
name.i – preprocessed source files

name.s – assembly files

name.o – object files

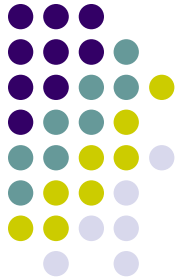
name – executable file

Preprocessor



- The first step in compilation
- Preprocessing changes the source code before the actual compilation begins
- It is a powerful tool. Practically a language within a language.
- Can be used (and misused) for different things, but you should try not to overuse it!
- Some of the things preprocessing directives are used for:
  - Header inclusion (module interface)
  - Literal naming
  - Quick “calls” of small functions
  - Conditional compiling

Result of preprocessing



- Many compiler offer possibility to print out how preprocessed source code looks like.
- It can be very useful for correct understanding of the code and debugging of preprocessing.
- In GCC switch –E tells compiler to store preprocessed code:

```
gcc –E file_name.c –o file_name.i
```

- Preprocessed text will be stored in file_name.i

app.h :

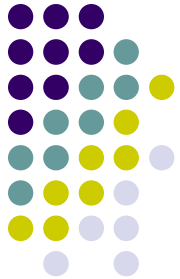
```
int extfunc(int a);
```

app.c:

```
#include "app.h"
#define MAX 30

int func(int a)
{
    if (a < extfunc(a>>2))
        return -1;
#ifdef LIMIT
    if (a > MAX)
        return 1;
#endif
    return 0
}
```

```
# 1 "app.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "app.c"

# 1 "app.h" 1
int extfunc(int a);
# 3 "app.c" 2


int func(int a)
{
    if (a < extfunc(a>>2))
        return -1;



    return 0
}
```

#include 1/2



- Probably the most important and mostly used preprocessing directive. It is hard to imagine even slightly more complex project without it.
- Usage of #include directive:

```
#include <stdio.h>
#include “app.h”
```

- In place of include directive, after preprocessing there will be copied text from the file specified as the directive’s parameter,.
- The file doesn’t have to be a header (with .h extension) but any textual file. Header as a concept is only programming convention, not something imposed by preprocessor.
- If file path in given inside these <>, then compiler will first look for the file in system folders. This way it is easy to avoid accidental shadowing of a system header by some user header.
- GCC by default look in these folders:

```
/usr/local/include
libdir/gcc/target/version/include
/usr/target/include
/usr/include
```

#include 2/2

- When file path is given in quotes, the compiler first looks in the user folders.
- Compiler usually provide mechanism for user to add include paths. In GCC, and in may others, this is command line switch used for that: **-Ifolder_name**

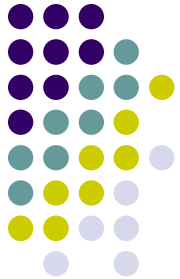
```
gcc -c -I../inc -I../libinc ../src/app.c -o app.o
```

- All paths se by -I switch are going to be searched before system paths, in case of quotes.
- It is even possible to tell GCC not to use standard header paths at all, by this switch: **–nostdinc**

```
gcc –nostdinc ../src/app.c
```

- In case of using quotes, this is the order of looking:
  - Path on which .c is
  - Path that are set by–I switch
  - Standard paths (unless –nostdinc is on)

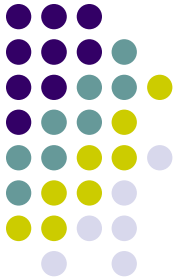
#define - Macros



- Two types of macros:
  - Macro-objects
  - Macro-functions
- Syntax:

```
Macro-objects:
#define MACRO_NAME [replacement token list]

Macro-functions:
#define MACRO_NAME(arg1, arg2, ...) [replacement token list]
```

- Wherever MACRO_NAME is encountered in the code, it will be replaced, by preprocessor, with the text defined by „replacement token list”.
- „Replacement token list” can be empty
- Usage of macro-functions has to have round brackets.
- It is common to write macros with all capital letters, so that they would be visually different then regular identifiers.



Macro-objects

- Macro-objects are often used for naming literals.
- Example:

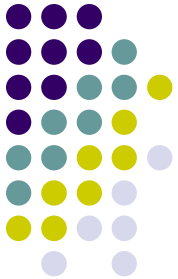
```
#define DELAY 50
```

- Naming literals is a good practice for several reasons:
  - Name can better describe meaning of a literal
  - If a need for changing literal comes up, and that literal is used in several places in code, it is easier just to change in one place
  - More clearly points out usage of a literal

```
for (i = 0; i < 5000; i++)
{
    /* loop body */
    usleep(5000);
}
```

```
for (i = 0; i < NUM_ITER; i++)
{
    /* loop body */
    usleep(DELAY);
}
```

- C99 standard introduced “const” qualifier, and it is better to use that.



Macro-functions

- Macros can be parametrized

```
#define MEET(X) I am X.
MEET(Djura)
I am Djura.
```

```
#define RADTODEG(X) ((X) * 57.29578)
```

- Macro-functions are not real functions and no instructions are spent for them.
- For macro-functions there mustn't be a space between name and round brackets.

```
#define RADTODEG (X) ((X) * 57.29578)
RADTODEG(2)
```

Will be reduced to:

```
(X) ((X) * 57.29578)(2)
```

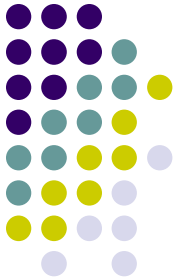
not to:

```
((2) * 57.29578)
```

- This applies only to definition, not for call

```
RADTODEG  (90) is the same as  RADTODEG(90)
```

Macros and operation priority



- Macro-functions are not real functions!
- They only define text transformation.
- Here is one typical problem that can happen if you apply logic related to regular functions on macro-functions.

```
#define RADTODEG(X) X * 57.29578
```

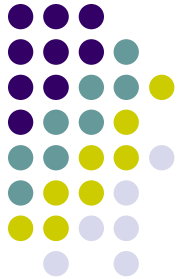
```
C / RADTODEG(A+B)

Will be reduced to:
C / A+B * 57.29578
```

- Multiplication has priority over additions, so in this case it leads to unexpected results.
- Solution is to place every parameter in braces, including the whole macro definition.

```
#define RADTODEG(X) ((X) * 57.29578)
```

```
C / RADTODEG(A+B)

Will be reduced to:
C / ((A+B) * 57.29578)
```

Macro-functions and side effects



- Evaluation of parameter side-effects can be very different between regular functions and macro functions.
- Example:

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

- Let us use the above macro-function in the following way:

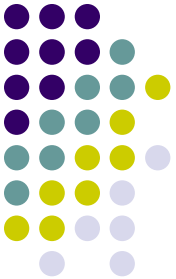
```
MIN(++x, y)
```

Will be reduced to:

```
((++x)>(y) ? (y) : (++x))
```

- If variable **x** is less or equal to **y**, then **x** will be incremented two times!
- In case of a regular function with the same body, **x** would be incremented only once.

; and macros



- It is usually not good practice to place ; at the end of macro definition.
- For example:

```
#define MIN(a,b) ((a)>(b) ? (b) : (a));
```

```
if (MIN(x,y) > 0)

Will be reduced to:
if ((x)>(y) ? (y) : (x)); > 0) /* SYNTAX ERROR */
```

- Or:

```
#define DEBUG(msg) printf(msg);
```

```
if (ret == 0)
   DEBUG("Success");
else
   DEBUG("Failed");

Биће сведено на:
if (ret == 0)
   printf("Success");; /* two statements */
else                   /* else not associated with if  */
   printf("Failed");;
```

inline functions

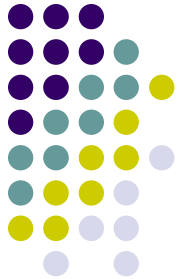
- For quick “calls” to small functions it is probably the best to use inline functions, which are introduced by C99 standard. If we declare a function to be ”static inline” then it will be in-lined at every call location, and definition will not even be generated. So, like a function-macro, only clearer and safer.

```
inline int min(int a, int b) {
    return (a > b) ? b : a;
}
```

- Inline function definitions can safely be placed in header files (by default linkage of inline function is internal)

- Two drawbacks:
  1. **inline** is just a suggestion, so compiler might not actually inline (although they rarely disobey).
  2. Inline functions are not type generic (although you can provide different functions for several different types and then use **_Generic** to select the proper version based on the argument type).

#if 1/3



- Enables different compiling, depending on some parameters.
- The following directives are used for that:
  - #if, #ifdef, #ifndef, #else, #elif
- All blocks started with #if, #ifdef or #ifndef have to ended with #endif

```
syntax:

#if expression
    text
#endif
```

- Expression is C expression, but with the following elements:
  - Only integer and character literals can be used – because it must be possible to evaluate the expression in compile time
  - Operators for: addition/subtraction, multiplication/division, bitwise operators, shift operators, comparisons, logical operators
  - Macros

#if 2/3

- Example

```
#if DEBUG > 2
    printf("Debug message\n");
#endif
```

- Only integers (and character literals, which are actually also integer literals):

```
#if DEBUG == "on" /* ERROR */
    printf("Debug message 1\n");
#endif

#if DEBUG > 2.0 /* ERROR */
    printf("Debug message 2\n");
#endif
```

- Can be used for quickly turning off (or on) some part of the code.

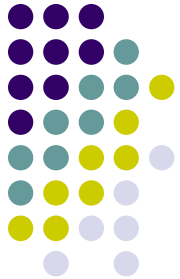
```
#if 0
    /* code */
#endif
```

- Only as a temporary solution.

- There is usually a way to define symbols from outside, through compiler switches.

```
gcc –D<b>DJURA</b> –D<b>DEBUG</b> –D<b>PERA</b>=14 –D<b>MILE</b>=(PERA-2) file_name.c
```

#if 3/3



- Another usage of conditional compilation is to avoid multiple inclusion of the same header.
- Included headers can include other headers, so it is quite hard to control what will be included and how many times.
- For that purpose, these directives are very useful: #ifndef и #define

```
#ifndef _FILE_NAME_H
#define _FILE_NAME_H

/* code */

#endif
```

- Similar construction can be used for avoiding multiple macro definitions

```
#ifndef NULL
#define NULL (void*)0
#endif
```

Token concatenation

- With operator **##** you can concatenate two tokens.
- Example:

```
#define DESCRIPTOR_FIELD(field) struct1.union1.m_##field
```

```
DESCRIPTOR_FIELD(pera)
```

Will be reduced to:

```
struct1.union1.m_pera
```

```
#define STR_AB djura##pera

x = STR_AB; -> x = djurapera;
```

but

```
x = djura##pera -> error
```

Stringifying tokens

- It is possible to convert a token to a string literal.
- It is useful if you want to print out a token.
- Operator # is used as unary operator.
- Example:

```
#define TOKEN(token) printf(#token " = %d\n", token)
```

```
TOKEN (x+y);

Will be reduced to:
printf("x+y" " = %d\n", x+y)
```

```
#define STR_T #djura

printf(STR_T); -> printf("djura");
but
printf(#djura); -> error!
```

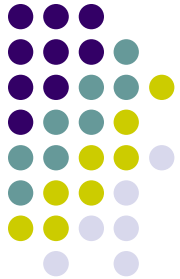
Predefined macros

- List of predefined macros:

Macro name	Description
__DATE__	Compilation date
__LINE__	Code line
__FILE__	File name
__TIME__	Compilation time
__STDC__	Data about supported standard

- Not a macro, but related: __func__ (look it up in the standard)

#error and #warning



- Using these directive it is possible to make compiler report a custom error or warning.

```
#error "Error message"
```

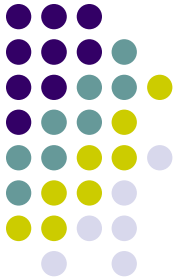
- Prints out the text, and halts compilation.

```
#warning "Warning message"
```

- Prints out the text as a warning, and continues compilation. This directive is not part of the standard, but almost all compiler support it.
- Example:

```
#ifdef  _WIN32
  /*  _WIN32 specific code */
#elif  defined ( linux )
  /* linux specific code */
  #warning "Linux version not fully supported"
#else
  #error "Not supported OS"
#endif
```

#pragma



- This directive's purpose is to serve as a channel for providing additional information to the compiler (information that can not be passed through language itself).
- Therefore, pragamas are mostly compiler specific, and it is up to copiler to define syntax and semantic of a pragma.

```
#pragma anything can go here: symbols, numbers, strings...
```

- There are only three standard pragmas. They are all related to floating point and complex numbers, and they have this format:

```
#pragma STDC *
```

- All the other pragmas that you encounter are platform/compiler specific.
- Pragma can apply to the whole file (wherever it is defined); just from its usage until the end of file (or until the line where another pragma nullifies); only on the next statement, or block, or line, etc.

```
#pragma once
#pragma pack(1)
#pragma GCC unroll n
```

- There is also **_Pragma** operator. The same meaning, but it allows pragma to be a result of a macro preprocessing.