Maciej Sobieraj

Lecture 13

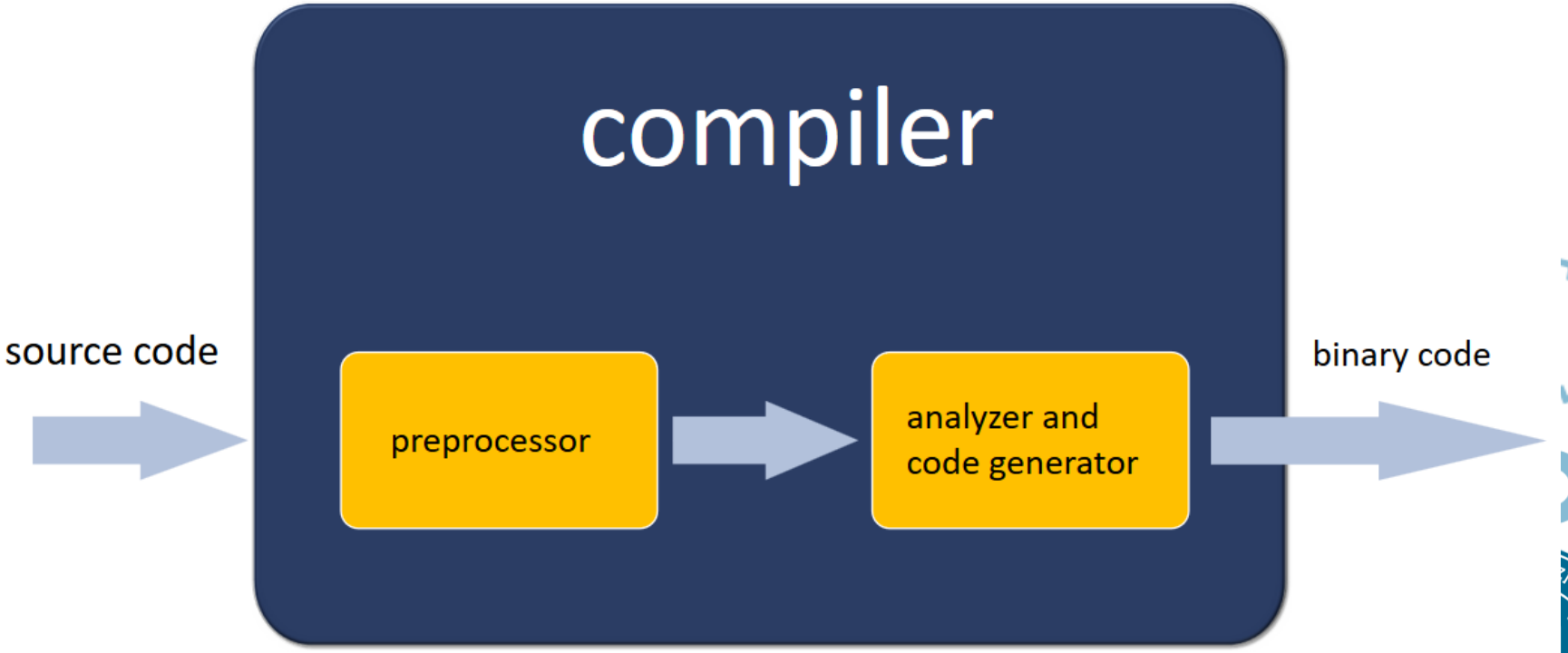
Outline

1. Preprocessor and declarations
    1. **Preprocessor: absolute basics**
    2. Preprocessor: the #include directive
    3. Preprocessor: the #define directive
    4. Preprocessor: the parameterized #define directive
    5. Preprocessor: the third variant of the #define and #undef directives
    6. Preprocessor: predefined identifiers
2. Quiz

The role of preprocessing

- The preprocessor is responsible for the **initial processing** of the program.
- the effects of preprocessing are **visible only to the compiler.**
- The form of your program, when it leaves the preprocessor, disappears at the same time the compiler exits.
- This means that the preprocessed source code is **not stored** in any way.

The role of preprocessing



The role of preprocessing

- The *gcc* compiler may be launched directly from the command line.
- If you want to know what your source code looks like after preprocessing, you'll need to use the *-E* option.

```
gcc -E prog.c
```

- The preprocessed source code will be sent to the standard output and all modifications ma by the preprocessor will be marked in a spec way.

The role of preprocessing

- If you want to preserve the preprocessed text for further analysis, you can **redirect** the standard output to a file and view it in your favorite text editor.

```
gcc -E prog.c > outputfilename
```

The role of preprocessing

- Some general principles:
  - the preprocessor directives **always begin** with the # which must be the **first visible character** in the line;
  - if the preprocessor directive doesn't fit on one line and needs to be split, you **must put the \ character** in the place where the directive is broken, e.g:
    - #include \
      <stdio.h>
  - any preprocessor directive placed in a particular file acts only inside this file and nowhere else.

Outline

1. Preprocessor and declarations
    1. Preprocessor: absolute basics
    2. **Preprocessor: the #include directive**
    3. Preprocessor: the #define directive
    4. Preprocessor: the parameterized #define directive
    5. Preprocessor: the third variant of the #define and #undef directives
    6. Preprocessor: predefined identifiers
2. Quiz

Including a file

- The *#include* directive works as follows:
  - the preprocessor looks for a **file named *filename***, but if you enclose the name inside quotes, it searches for that file in the same directory as the file containing the directive; otherwise it searches for the file in the compiler's default directory;
  - if the file is not found, the preprocessor reports an error; otherwise, the *#include* directive is **replaced** with the content of the included file;
  - …

Including a file

- The *#include* directive works as follows:
  - …
  - the included file **may contain** an *#include* directive, too; in addition, a particular compiler may impose a certain limit on the number of nested inclusions;
  - you may use the *#include* directive in any part of the source file, not only at the beginning.

```
#include <filename>

#include "filename"
```

Including a file

- Let's assume that we have two source files. The first one is named *src1.c* and contains the following text:
  - ```
    int main(void) {
        #include "src2.c"
    }
    ```
- The second one is named *src2.c* and looks like this:
  - ```
    int i = 0;
    return i;
    ```

Including a file

- When you start the *gcc* compiler with the following command:
  - gcc -E src1.c
- you see the following text displayed on the screen:
- the lines marked with the # character are used internally by the compiler,

```
# 1 "src1.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "src1.c"
int main(void) {

# 1 "src2.c" 1
int i = 0;

return i;
# 4 "src1.c" 2
}
```

Outline

1. Preprocessor and declarations
    1. Preprocessor: absolute basics
    2. Preprocessor: the #include directive
    3. **Preprocessor: the #define directive**
    4. Preprocessor: the parameterized #define directive
    5. Preprocessor: the third variant of the #define and #undef directives
    6. Preprocessor: predefined identifiers
2. Quiz

Three forms of the directive

- The *#define* directive may take one of the following three forms

```
#define identifier text

#define identifier(parameter_list) text

#define identifier
```

Three forms of the directive

- Let’s explain the first form of the directive now. Its operation is as follows:
  - the preprocessor **remembers the *identifier*** and the ***text* associated with it**;
  - from this moment on, the preprocessor analyses the source code, **replacing** any occurrence of the *identifier* with the associated *text*;
  - the directive itself **won't appear** in the preprocessed source code.

```
#define identifier text
```

#define: simple substitution

Source code:

```
#define PI 3.1415

int main(void) {
    float s = 2 * PI * 10.0;
    return 0;
}
```

- Note: *PI* is an identifier, while *3.1415* is text. The preprocessed code will take the following form

Preprocessed code:

```
int main(void) {
    float s = 2 * 3.1415 * 10.0;
    return 0;
}
```

#define: simple substitution

- The identifier used inside the *#define* directive (such as *PI*) is often called *a* **macro** and the process of replacing it with corresponding text is called a **macro substitution**.
- You usually write the macro names in capital letters, allowing them to be distinguished from ordinary variable names.
- This is not an absolute rule, but fairly widely recognized and used.

#define: simple substitution

- This form of the directive is often used to improve program readability and to make it easier to modify.

```
#include <stdio.h>

int main(void) {
        int arr[100], i, sum = 0;
        for(i = 0; i < 100; i++)
                arr[i] = 2 * i;
        for(i = 0; i < 100; i++)
                sum += arr[i];
        for(i = 0; i < 100; i ++)
                printf("%d\n", arr[i]);
        printf("%d\n", sum);
        return 0;
}
```

#define: simple substitution

```c
#include <stdio.h>

#define  SIZE      100

int main(void) {
      int arr[SIZE], i, sum = 0;
      for(i = 0; i < SIZE; i++)
            arr[i] = 2 * i;
      for(i = 0; i < SIZE; i++)
            sum += arr[i];
      for(i = 0; i < SIZE; i ++)
            printf("%d\n", arr[i]);
      printf("%d\n", sum);
      return 0;
}
```

#define: disadvantages and perils

```
#define  SIZE            4+4
int main(void) {
        int i;
        i = 2 * SIZE;
        printf("%d\n",i);
        return 0;
}
```

#define: disadvantages and perils

- At first glance, it would seem that the value assigned to the *i* variable is 16 (actually: 2 * 8).
- Unfortunately, nothing could be further from the truth. Don't forget that the preprocessor doesn’t use the **value** of the identifier, but only **replaces** it with the associated **text**.

#define: predefined identifiers

- Many of the identifiers we've used so far (for example, the NULL symbol or the EOF symbol) are actually **macros**.
- If we take a look into the *stdio.h* and *stdlib.h* header files, we find (among many, many others) the following *#define* directives

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

#define  EOF         (-1)

#define  SEEK_SET    0
#define  SEEK_CUR    1
#define  SEEK_END    2
```

Outline

1. Preprocessor and declarations
    1. Preprocessor: absolute basics
    2. Preprocessor: the #include directive
    3. Preprocessor: the #define directive
    4. **Preprocessor: the parameterized #define directive**
    5. Preprocessor: the third variant of the #define and #undef directives
    6. Preprocessor: predefined identifiers
2. Quiz

The second form of the #define directive

```
#define identifier(parameter_list) text
```

- The *parameter_list* element should consist of different **pairwise identifiers** (at least one), separated by **commas**.
- A macro defined in this way is called a **macro with parameters**, or a **parameterized macro**.

The second form of the #define directive

- When the preprocessor meets an identifier identical to the one defined as a macro and there's a parameter list after it, it takes the following actions:
  - each macro parameter in the text is **replaced** by the argument specified in the source code;
  - the macro identifier, along with the parameters list, is **replaced** by the text composed in the first step;
  - the directive itself **won't appear** in the preprocessed source code.

Parameterizing the #define directive

- The first step of the macro substitution will cause the x parameter to be replaced with the macro's argument (i.e. "*length*").

Source code:

```
#define SQR(x)     (x * x)
float f = SQR(length);
```

Preprocessed code:

```
float f = (length * length);
```

A macro is not a function!

- Note: a **macro is not a function**. It only looks like one.

Source code:

```
#define SQR(x)     (x * x)
float f = SQR(length + 1)
```

Preprocessed code:

```
float f = (length + 1 * length + 1)
```

A macro is not a function!

- If we rewrite the macro in the following way:
  - #define SQR (X) ((X) * (X))
- it works as expected:
  - **float** f = ((length + 1) * (length + 1))

A macro is not a function!

Source code:

```
#define SQR(x)     ((x) * (x))
float f = SQR(length++)
```

Preprocessed code:

```
float f = ((length++) * (length++))
```

A macro is not a function!

- If SQR is a function:
  - it returns the value of (*length * length*);
  - the *length* variable is increased by 1.
- Since SQR is a macro, the f variable field will be assigned with the value of the following expression:
  - ((length++) * (length++))

Some examples of parameterized macros

Source code:

```
#define  MAX(X,Y)      (((X) > (Y)) ? (X) : (Y))

int main(void) {
      int var1 = 100, var2 = 200, var;
      float v1 = -1.0, v2 = 1.0, v;

      var = MAX(var1,var2);
      v = MAX(v1,v2);

      return 0;
}
```

Some examples of parameterized macros

Preprocessed code:

```
int main(void) {
    int var1 = 100, var2 = 200, var;
    float v1 = -1.0, v2 = 1.0, v;

    var = (((var1) > (var2)) ? (var1) : (var2));
    v = (((v1) > (v2)) ? (v1) : (v2));

    return 0;
}
```

Some examples of parameterized macros

Source code:

```
#define MAX(X,Y)    (((X) > (Y)) ? (X):(Y))
#define MAX3(X,Y,Z) (MAX((MAX((X),(Y))),(MAX((Y),(Z)))))

int main(void) {
      int a = 1, b = 2, c = 3, w;
      w = MAX3(a + 1,b - 1,2 * c);
}
```

Some examples of parameterized macros

Preprocessed code:

```
int main(void) {
int a = 1, b = 2, c = 3, w;
w = (((((((((a + 1)) > ((b – 1))) ? ((a + 1)) : ((b – 1))))) > ((((((b – 1)) > ((2 * c))) ? ((b – 1)) : ((2 * c)))))) ?

    ((((((a + 1)) > ((b – 1))) ? ((a + 1)) : ((b – 1))))) : ((((((b – 1)) > ((2 * c))) ? ((b – 1)) : ((2 * c)))))));
    }
```

Outline

1. Preprocessor and declarations
    1. Preprocessor: absolute basics
    2. Preprocessor: the #include directive
    3. Preprocessor: the #define directive
    4. Preprocessor: the parameterized #define directive
    5. **Preprocessor: the third variant of the #define and #undef directives**
    6. Preprocessor: predefined identifiers
2. Quiz

The third variant of the #define directive

#define identifier

- The directive affects preprocessor operations in the following way:
  - it causes the preprocessor to **assume** that the *identifier* is a **defined** identifier (i.e. known to the compiler);
  - the directive itself won't appear in the preprocessed source code;
  - the source code **is not changed** in any way.

The #undef directive

- The *#undef* directive **cancels** the effects of the selected *#define* directive and takes the following form:

```
#undef identifier
```

- From the moment the *#undef* directive is used, the *identifier* is not defined.

The #undef directive

```
int add(int x) {
        return x + 1;
}

int main(void) {
        int i = 100;
        i = add(i);
#define add(x)    (2 * (x))
        i = add(i);
#undef add
        i = add(i);
        printf("%d",i);
        return 0;
}
```

The #undef directive

- Before the preprocessor sees the first *#define* directive, *add* is undefined (from the preprocessor's perspective), so the appearance of the phrase:
  - i = add(i);
- is treated as a **function invocation** and doesn't elicit any reaction from the preprocessor.

The #undef directive

- The preprocessed source code will look as follows:
  - ```
    int add(int x) {
            return x + 1;
    }
    int main(void) {
            int i = 100;
            i = add(i);
            i = (2 * ( i )) ;
            i = add(i);
            printf("%d",i);
            return 0;
    }
    ```


    ```

Outline

1. Preprocessor and declarations
    1. Preprocessor: absolute basics
    2. Preprocessor: the #include directive
    3. Preprocessor: the #define directive
    4. Preprocessor: the parameterized #define directive
    5. Preprocessor: the third variant of the #define and #undef directives
    6. **Preprocessor: predefined identifiers**
2. Quiz

The __LINE__ identifier

- A number of identifiers are **defined by the preprocessor itself**.
- Wherever you use the *__LINE__* it is replaced by an **integer literal equal to the line number**, where the symbol appears.
- This means that the symbol will have a different value in every line of your source code.

The __LINE__ identifier

- Note the empty lines intentionally placed inside the text.

```
#include <stdio.h>

int main(void) {
    printf("this is line #%d\n", __LINE__);

    printf("this is line #%d\n", __LINE__);



    printf("this is line #%d\n", __LINE__);
    return 0;
}
```

The __LINE__ identifier

- This code, when passed through the preprocessor, will take the following form:
  - ```
    #include <stdio.h>

    int main(void) {
        printf("this is line #%d\n", 4);

        printf("this is line #%d\n", 6);



        printf("this is line #%d\n", 10);
        return 0;
    }
    ```


    ```

The __FILE__ identifier

- The *__FILE__* identifier is always replaced by a string literal containing the **name of the source file** in which the identifier was used.

```
#include <stdio.h>

int main(void) {
	puts("Hello from the source file named "__FILE__);
	return 0;
}
```

The __FILE__ identifier

- Assuming that the code was placed in a file named *filesym.c*, it will take the following form after it passes through the preprocessor:
  - #include <stdio.h>

```
int main(void) {
    puts("Hello from the source file named ""filesym.c");
    return 0;
}
```

- The same code, compiled and run, will emit the following text to the standard output:
  - Hello from the source file named filesym.c

The __FILE__ identifier

- There's a rule in the "C" language that says that a string literal can be broken at any point with the " character, and then any number (including zero) of white characters may appear; after which, the literal may be resumed with another " character.
- oreover, a source code written like this:
  - ```
    int main(void) {
            puts("Hello from the source file named "
            __FILE__);
            return 0;
        }
    ```


    ```

The __DATE__ identifier

- The __DATE__ identifier is always replaced by a string literal containing **text denoting the day** the source file was compiled.

```
#include <stdio.h>

int main(void) {
	puts("The program was successfully compiled on " __DATE__);
	return 0;
}
```

The __DATE__ identifier

- his code, after passing through the preprocessor, may look as follows:
  - #include <stdio.h>

```
int main(void) {
    puts("The program was successfully compiled on " "Aug 22 2012");
     return 0;
}
```

- That means that this code, when compiled and run, will emit the following text to the standard output:
  - The program was successfully compiled on Aug 22 2012

The __TIME__ identifier

- The __TIME__ identifier is always replaced by a string literal containing **text denoting the time** (hours, minutes, seconds) the source file was compiled.

```
#include <stdio.h>

int main(void) {
	puts("I was compiled at " __TIME__);
	return 0;
}
```

The __TIME__ identifier

- This code, after passing through the preprocessor, may look as follows:
  - #include <stdio.h>

```
int main(void) {
    puts("I was compiled at " "12:13:23");
    return 0;
}
```

- And this code, when compiled and run, will emit the following text to the standard output:
  - I was compiled at 12:13:23

The __STDC__ identifier

- The *__STDC__* identifier (as in **St**andar**d C**) is defined if and only if the following statement is true:
  - *the compiler is operating in compliance with the ANSI "C" standard*
- When *ANSI* mode is on, it means that the compiler honors only the language elements described in the standard documents, and no extensions or limitations are applied.
- If the compiler isn't working in ANSI mode, th symbol is not defined.

Outline

1. Preprocessor and declarations
    1. Preprocessor: absolute basics
    2. Preprocessor: the #include directive
    3. Preprocessor: the #define directive
    4. Preprocessor: the parameterized #define directive
    5. Preprocessor: the third variant of the #define and #undef directives
    6. Preprocessor: predefined identifiers
2. **Quiz**

Quiz

What happens when you compile and run the following program?

```
#include <stdio.h>
#define X 2
#define Y X*X
#define Z Y+Y
#define MINUS 2-4
int main(void) {
   int i = Z;
   int j = i * MINUS;
   int k = i + j;
   printf("%d",k);
   return 0;
}
```

- ○ the program outputs 20
- ○ the program outputs 30
- ○ the program outputs 40

Quiz

What happens when you compile and run the following program?

```
#include <stdio.h>
#define F1(X)     X*X
#define F2(X)     ((X)*(X))
#define F3(X)     ((X)*X)
int main(void) {
   int i = 1;
   int j = 2;
   int k = 3;
   int s;
   s = F1(i + j) + F2(i - j) + F3(i + k);
   printf("%d",s);
   return 0;
}
```

- ○ the program outputs 16
- ○ the program outputs 13
- ○ the program outputs 10

Quiz

What happens when you compile and run the following program?

```
#include <stdio.h>
int main(void) {
   int X = 200;
   int a = X;
   a += X;
#define X 200
   a += X;
#undef X
   printf("%d",a);
   return 0;
}
```

- the program outputs 300
- the program outputs 500
- the program outputs 600

Quiz

What happens when you compile and run the following program?

```
#include <stdio.h>
int X = 0;
#define X 100;
int fun1(void) {
   return X;
}
#undef X
int fun2(void) {
   return X;
}
int main(void) {
   int s;
   s = fun1() + fun2();
   printf("%d",s);
   return 0;
}
```

- ○ the program outputs 200
- ○ the program outputs 100
- ○ the program outputs 300