Lecture 11 - PREPROCESSOR

Preprocessor is called in the first phase of the program translation; before starting the proper compilation.

Preprocessor processes the source text of the program !!!

Inserting files

One or two lines of the following kind often appear at the beginning of the program:

```
#include <name_of_file>
or
#include "name_of_file"
```

Those lines mean that the content of the file determined by `nazwa_pliku` will be inserted at the position of the **include** statement appearance.
`<name_of_file>` means that the file will be searched according to the rules holding in the implementation.
`"name_of_file"` means that search of the file is started there where the source program is found and if the it doesn't exist in that place, then the search is continued according to the rules holding in the implementation.

Restrictions on the use of the first form:

- Character `>` and the new line character could not appear in the `name_of_file`.
- Result of that statement is not determined if any of the following characters appear: `"` `'` `\` or pair of characters `/*` .

Restrictions on the use of the second form:

- Result of using characters `'` `\` and pair of characters `/*` is still not determined.
- character `>` is allowed.

Use of a line of the following form is also allowed:

```
#include string_of_characters
```

`string_of_characters` is then expanded according to the usual rules and at the end it should lead to one of the forms: `<...>` albo `"..."`.

Application of `#include` is a suggested way of preparing declarations for a large program. It guarantees identity of the *definitions and declarations* of variables and functions.

Remark: Inclusion of files may be nested.

Macroexpansions – replacement of a name by string of characters.

Scope of the statement - from the place of appearance till the end of the file.

Control line of the form

```
#define identifier string_of_characters
```

instructs preprocessor to replace further appearances of the identifier by the indicated string of characters. Blanks surrounding string of characters are omitted.

Remark Next appearance of `#define` with the same identifier is treated as an error if the strings of characters are in both cases identical (all white spots dividing strings of characters are treated as equivalent).

```
Examples:
1)
#define YES 1
2)
#define then
#define begin   {
#define end    ;}
```

Afterwards we may write as follows:

```
if (i>0) then
  begin
    a=1;
    b=2;
  end
```

Macro with arguments

```
#define identifier(list_of_identifiers)   \
                                    string_of_characters
```

Example:

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

Parantheses in the definition are important since for instance:

```
x=max(p+q,r+s);
                shall be replaced by
x=((p+q)>(r+s)?(p+q):(r+s);
```

Several rules:

- definition may make use of the previous definitions (nested definition),
- macroexpansion does not take place in strings of characters surrounded by the quotation marks, e.g.

  ```
  printf("YES");
  ```

  prevents any replacement of YES.

Example 2:

```
/*  macro reversing the places of arguments  */
#define SWAP(x,y)   {                       \
                      double tmp;           \
                      tmp=x;                \
                      x=y;                  \
                      y=tmp;                \
                    }
```

Example 3:

```
#define APPEND(name,f,thing)                 \
           {                                 \
             FILE *out;                      \
             out=fopen("a:\\kat\\"#name,"a");\
             fprintf(out,f,thing);           \
             fclose(out);                    \
           }
```

Remark: Expressions having some aside effects function calls should be avoided, since after expansion of the macrodefinition text may be included several times.

Examples:

```
#define max(x,y) ((x)>(y)?(x):(y))

maximum=max(++a,10);
    /* result will be either 10 or a+2  */
    /* a will be increased by  1 or 2   */

maximum=max(fgetc(file),maximum);
    /* one can find in that way the neighbour
       of the highest character in the file */

maximum=max(rand(),maximum);
    /*   !!!!!!                          */
```

Remark: Semicolon character should be used with care within macrodefinitions.

Control line of the form

```
#undef identifier
```

orders the preprocessor to forget the identifier definition. Application of the `#undef` statement to an undefined identifier is not treated as an error.

Characters # and

If a paarameter is preceded by the character `#` then the parameter identifier together with the `#` character shall be replaced by an appropriate argument surrounded by the quotation marks `"` .

Each character `"` and `\` appearing at the beginning, in the middle or at the end of the string constants and strings of characters creating the argument shall be preceded by the character `\` .

If operator `##` appears within the string of characters defining macro then after replacement of parameters by the strings of characters – operator `##` togther with surrounding it white spots shall be removed.

If the string of characters created in that way is incorrect or its execution depends on the order, then the result is undefined.

Independently of the macrodefinition form the replacing string of characters is many times searched for in search for other identifiers defined in that way. However if an identifier replaced already in the current expansion appears again in the next search, it will remain unchanged in the expanded text.

Macrodefinitions mechanism is useful to define "important constants"

Example:

```
#define ABSDIFF(a,b)  ((a)>(b)?(a)-(b):(b)-(a))
```

Defines macro returning absolute value of the difference of its arguments.
In contrary to the functions the returning value may be of any type.

```
#define tempfile(dir) #dir "/%s"
```

Call `tempfile(/usr/tmp)` gives as the result

```
"/usr/tmp" "/%s"
```

that next shall be sticked together into one string of characters. In the presence of the definition

```
#define cat(x,y)  x ## y
```

call

```
cat(var,123)
```

produces string of characters

```
var123
```

Result of call

```
cat(cat(1,2),3)
```

is not determined; presence of operartor `##` prevents expansion of arguments of the internal call.

As the result appears `cat(1,2)3` , at which string of `)3` (result of sticking the last string from the first argument with the first string from the second argument is not correct.

After introduction of a macro of the second level

```
#define xcat(x,y) cat(x,y)
```

all works more appropriately: *xcat(xcat(1,2),3)* indeed creates string of characters *123*, since in the xcat itself operator `##` does not appear.

Similarly `ABSDIFF(ABSDIFF(a,b),c)` creates result fully expanded consistently with the expectations.

Conditional compilation

Parts of program may be compiled conditionally accordingly to the below schematic syntax:

```
Conditional compilation:
      line-if text of the part -elif part-else  #endif

      line_if:
         #if constant-expression
         #ifdef identifier
         #ifndef identifier

      parts el-if:
         line-elif text
         parts-elif

      line-elif:
         #elif constant expression

      part-else
         line_else text

      part-else:
         line-else text

      line-else:
         #else
```

Several rules:

- Constant expressions appearing in `#if` and `#elif` are calculated consecutively till the appearance of an expression with nonzero value.
- Text following expressions with zero value is omitted during compilation.
- Text appearing after expressions with positive value is included into the program.
- After finding an expression with positive value and its servicing, the subsequent lines `#elif` or `#else` sąre omitted together with their texts.
- If all expressions are equal to zero and there exists the `#else` , then text following `#else` is serviced.

- Text appearing in inactive branches is ignored, however it is checked first, whether there doesn't exist nested conditional constructions.

Constant expressions in `#if` and `#elif` are subject to a standard macrodefinitions.

Expressions of the form:

```
    defined identifier
or
    defined ( identifier )
```

take before expansion the following values:

1L - if the identifier is defined in the preprocessor

0L - if it is not defined

All identifiers left after macroexpansions are set to `0L` , and the whole arithmetic on constants is carried out on long integer or unsigned long integer numbers.

There exist the following restrictions on resulting constant expressions:

- they must be integer
- they could not contain operators *sizeof*, casting and the enumeration constants.

Control lines of the form:

```
#ifdef identifier
#indef identifier
```

are equivalent to:

```
#if defined identifier
#if ! defined identifier
```

Examples:

```
1)  /* Compiling program in several versions  */

#define DEMO
#ifdef DEMO
/*  code exclusively for demo version  */

 statements

#endif

#ifndef DEMO
/*  code exclusively for normal version  */
#endif

2)  /*  To prevent multiple expansion of the same
        piece of source code   */

    /*  beginning of the header file   windowws.h  */
    #ifndef WINDOWS
    #define WINDOWS

    /*  content of the file  */
    #endif
    /*  End of file  windows */
```

Lines numbering

Line numbering is introduced to fulfill the requirements of other preprocessors generating programs in C language. It is realized by writing line having one of the following forms:

```
#line constant "name_of_file"
#line constant
```

that instructs the compiler to assume for diagnostic purposes that the next source line will have number determined by the constant and that `name_of_file` will be the name of the current file.

Macrosses appearing in such lines are expanded before the statement interpretation.

Generating errors

Control line of the form

```
#error string_of_characters
```

instructs preprocessor to right diagnostic message containing the given `string_of_characters` .

Instruction *pragma*

Control line of the form

```
#pragma string_of_characters
```

instructs preprocessor to carry out some action depending on the implementation. Unknown action is ignored.

Empty preprocessor statement

Line containing only character `#` does not have any effect.

Names defined in preprocessor

Definitions of such names and **defined** operator (appearing in the preprocessor statements) could not be changed by the programmer.

- __LINE__ – decimal integer constant containing containing tha number of the current line of the program
- __FILE__ – constant string of characters containing the name of the translated file
- __DATE__ – constant string of characters containing the date of the program translation stored in format – `"Mmm dd rrrr"`
- __TIME__ – constant string of characters containing time of program translation stored in format – `"gg:mm:ss"`
- __STDC__ – constant *1*. The aim was that this identifier should be set equal to 1 only in implementations fitting the standard.