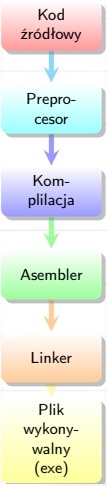

Preprocesor języka C: dyrektywy, makrodefinicje

wer. 9 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2021-04-22 07:38:56 +0200

Od kodu do programu wykonywalnego

1. Kod źródłowy
2. Preprocesor (gcc -E)
3. Kompilacja (konwersja do języka wewnętrznego) (gcc -S)
4. Assembler (tworzy pliki wynikowe) (gcc -c)
5. Konsolidowanie (przeglądanie bibliotek, budowa pliku wykonywalnego).
6. Uruchomienie
7. Debugowanie

Preprocesor

1. Pierwszym krokiem kompilacji jest przetwarzanie programu za pomocą **preprocesora**.

Preprocesor

1. Pierwszym krokiem kompilacji jest przetwarzanie programu za pomocą **preprocesora**.
2. Najczęściej stosowanymi poleceniami preprocesora są:

Preprocesor

1. Pierwszym krokiem kompilacji jest przetwarzanie programu za pomocą **preprocesora**.
2. Najczęściej stosowanymi poleceniami preprocesora są:
   - **#include**

Preprocesor

1. Pierwszym krokiem kompilacji jest przetwarzanie programu za pomocą **preprocesora**.
2. Najczęściej stosowanymi poleceniami preprocesora są:
   - **#include**
   - **#define**

Preprocesor

1. Pierwszym krokiem kompilacji jest przetwarzanie programu za pomocą **preprocesora**.
2. Najczęściej stosowanymi poleceniami preprocesora są:
   - **#include**
   - **#define**
   - polecenia kompilacji warunkowej (**#if**, **#ifdef**, **#ifndef**, **#endif**, **#else**, **#elif**)

Wstawianie plików

1. Każdy wiersz programu o postaci:

jest zastępowany zawartością pliku o wskazanej nazwie.

Wstawianie plików

1. Każdy wiersz programu o postaci:
   - **#include** "*nazwa-pliku*"

   jest zastępowany zawartością pliku o wskazanej nazwie.

Wstawianie plików

1. Każdy wiersz programu o postaci:
   - **#include** "*nazwa-pliku*"
   - **#include** <*nazwa-pliku*>

   jest zastępowany zawartością pliku o wskazanej nazwie.

Wstawianie plików

1. Każdy wiersz programu o postaci:
   - **#include** "*nazwa-pliku*"
   - **#include** <*nazwa-pliku*>

   jest zastępowany zawartością pliku o wskazanej nazwie.
2. Jeśli nazwa pliku ograniczona jest cudzysłowami — poszukiwanie pliku rozpoczyna się tam gdzie znaleziono plik źródłowy.

Wstawianie plików

1. Każdy wiersz programu o postaci:
   - **#include** "*nazwa-pliku*"
   - **#include** <*nazwa-pliku*>

   jest zastępowany zawartością pliku o wskazanej nazwie.
2. Jeśli nazwa pliku ograniczona jest cudzysłowami — poszukiwanie pliku rozpoczyna się tam gdzie znaleziono plik źródłowy.
3. Jeśli nazwa pliku ograniczona jest nawiasami kątowymi <, > plik szukany jest wśród plików „systemowych” (w miejscach zależnych od implementacji).

Wstawianie plików

1. Każdy wiersz programu o postaci:
   - **#include** "*nazwa-pliku*"
   - **#include** <*nazwa-pliku*>

   jest zastępowany zawartością pliku o wskazanej nazwie.
2. Jeśli nazwa pliku ograniczona jest cudzysłowami — poszukiwanie pliku rozpoczyna się tam gdzie znaleziono plik źródłowy.
3. Jeśli nazwa pliku ograniczona jest nawiasami kątowymi <, > plik szukany jest wśród plików „systemowych” (w miejscach zależnych od implementacji).
4. Plik włączany poleceniem **#include** może zawierać kolejne polecenia **#include**.

Standardowe pliki nagłówkowe I

ANSI C library header files

Ważne pliki nagłówkowe:

1. <assert.h> Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
2. <complex.h> A set of functions for manipulating complex numbers. (New with C99)
3. <ctype.h> Contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
4. <errno.h> For testing error codes reported by library functions.
5. <fenv.h> For controlling floating-point environment. (New with C99)

Standardowe pliki nagłówkowe II

ANSI C library header files

6. <**float**.h> Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX). 
7. <inttypes.h> For precise conversion between integer types. (New with C99)
8. <iso646.h> For programming in ISO 646 variant character sets. (New with NA1)
9. <limits.h> Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX). 

Standardowe pliki nagłówkowe III

ANSI C library header files

10. <locale.h> For setlocale() and related constants. This is used to choose an appropriate locale.
11. <math.h> For computing common mathematical functions 
12. <setjmp.h> Declares the macros setjmp and longjmp, which are used for non-local exits
13. <signal.h> For controlling various exceptional conditions 
14. <stdarg.h> For accessing a varying number of arguments passed to functions. 
15. <stdbool.h> For a boolean data type. (New with C99)
16. <stdint.h> For defining various integer types. (New with C99)
17. <stddef.h> For defining several useful types and macros.

Standardowe pliki nagłówkowe IV

ANSI C library header files

18. <stdio.h> Provides the core input and output capabilities of the C language. This file includes the venerable printf function.
19. <stdlib.h> For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
20. <string.h> For manipulating several kinds of strings.
21. <tgmath.h> For type-generic mathematical functions. (New with C99)
22. <time.h> For converting between various time and date formats.
23. <wchar.h> For manipulating wide streams and several kinds of strings using wide characters — key to supporting a range of languages. (New with NA1)
24. <wctype.h> For classifying wide characters. (New with NA1)

Makrorozwinięcia I

1. Makrorozwinięcia to definicje które pozwalają duży fragment tekstu zastąpić krótkim:

```
#define nazwa zastepujacy_tekst
```

2. Każde użycie ciągu znaków `nazwa` zostanie zamienione przez *zastepujacy_tekst*.
3. Definicja może korzystać z poprzednich definicji.
4. Makrorozwinięcia nie obowiązują wewnątrz stałych tekstowych (ograniczonych znakami cudzysłowów).
5. Makrorozwinięcia stosuje się do całych jednostek leksykalnych. Zatem jeżeli zdefiniowane jest coś takiego:

```
#define YES "Tak"
```

Makrorozwinięcia II

to każde wystąpienie YES zostanie zamienione na "Tak", ale nie będzie to dotyczyło

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

albo wystąpienia napisu „YESMAN”
A co będzie w przypadku

```
printf( YES );
```

Rzeczywiste przykłady

Z pliku math.h

```
# define M_PI           3.14159265358979323846  /* pi */
# define M_PI_2         1.57079632679489661923  /* pi/2 */
```

Z pliku stdlib.h

```
#define EXIT_FAILURE    1 /* Failing exit status.  */
#define EXIT_SUCCESS    0 /* Successful exit status.  */
```

Z pliku stdin.h

```
#define EOF (-1)
```

Makrorozwinięcia z parametrami I

1. Istnieje możliwość definiowania makr z argumentami — zastępujący tekst może być różny dla różnych wywołań:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Bardzo przypomina wywołanie funkcji, ale realizowane będzie w sposób następujący:

- Pojawienie się w tekście programu napisu

```
x = max(p+q, r+s);
```

zostanie rozwinięte jako

```
x = ((p+q) > (r+s) ? (p+q) : (r+s))
```

- Rozwiązanie takie ma wadę: wyrażenia obliczane są dwukrotnie, zatem w

```
z = max(i++, j++);
```

Makrorozwinięcia z parametrami II

wartości `i`, `j` zostaną zwiększone dwukrotnie!

2. W przypadku gdy makrorozwinięcia używane są do prowadzenia obliczeń, pamiętać należy o odpowiednim stosowaniu nawiasów.

```
#define square(x) x * x
```

tłumaczone jest rozsądnie w najprostszym przypadku `y = square(v);` na

```
y = v * v;
```

ale w przypadku `y = square(v + 1);` na

```
y = v + 1 * v + 1;
```

3. Poprawna definicja powinna wyglądać jakoś tak:

```
#define square(x) (x) * (x)
```

4. Uwaga: pomiędzy nazwą a nawiasem otwierającym nie może być odstępu!
5. Makra rozwijane są przed kompilacją!

Makrorozwinięcia z parametrami I

1. Jeżeli nazwę parametru w zastępującym tekście poprzedzimy znakiem # to cała kombinacja (parametr i znak) zostaną rozwinięte w ciąg znaków ograniczonych cudzysłowami.
2. Przyjmijmy, że mamy taką definicję:

```
#define drukuj(tekst) printf( #tekst )
```

to wywołanie

```
drukuj(ala ma kota);
```

jest rozwijane w

```
printf( "ala ma kota" );
```

3. Znaków ## można używać do sklejania parametrów formalnych. Makro

```
#define sklej(przod, tyl) przod ## tyl
```

Makrorozwinięcia z parametrami II

wywołane

```
sklej(a, 1)
```

zostanie rozwinięte jako a1

Makrorozwinięcia

Prosty przykład

```
#ifdef DEBUG
# include <stdio.h>
# define wypisz(x) do { printf("Zmienna " # x \
                                " = %d\n", x); } \
   while (0);
#else
# define wypisz(x)
#endif
```

Użycie

```
#define DEBUG
...
wypisz( a );
```

Odwoływanie makra

Jeżeli jakaś definicja przestaje być potrzebna można ją zlikwidować za pomocą polecenia

```
#undef
```

Na przykład tak:

```
#define DEBUG 1 /* Wydruki diagnostyczne */
/* ...          */
#ifdef DEBUG
/* ...          */
#endif
/* ...          */
#undef DEBUG
/* ...          */
```

Zmienne preprocesora

Wiele z tego zależy od implementacji, ale zazwyczaj zdefiniowane są następujące zmienne:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

1. __LINE__ i __FILE__ oznaczają, odpowiednio, numer linii źródłowej kompilowanego pliku i jego nazwę.

Zmienne preprocesora

Wiele z tego zależy od implementacji, ale zazwyczaj zdefiniowane są następujące zmienne:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

1. __LINE__ i __FILE__ oznaczają, odpowiednio, numer linii źródłowej kompilowanego pliku i jego nazwę.
2. Zmienna __DATE__ zawiera datę — precyzyjniej datę kiedy dokonywana jest kompilacja.

Zmienne preprocesora

Wiele z tego zależy od implementacji, ale zazwyczaj zdefiniowane są następujące zmienne:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

1. __LINE__ i __FILE__ oznaczają, odpowiednio, numer linii źródłowej kompilowanego pliku i jego nazwę.
2. Zmienna __DATE__ zawiera datę — precyzyjniej datę kiedy dokonywana jest kompilacja.
3. Zmienna __TIME__ zawiera czas — precyzyjniej czas kiedy dokonywana jest kompilacja.

Zmienne preprocesora

Wiele z tego zależy od implementacji, ale zazwyczaj zdefiniowane są następujące zmienne:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

1. __LINE__ i __FILE__ oznaczają, odpowiednio, numer linii źródłowej kompilowanego pliku i jego nazwę.
2. Zmienna __DATE__ zawiera datę — precyzyjniej datę kiedy dokonywana jest kompilacja.
3. Zmienna __TIME__ zawiera czas — precyzyjniej czas kiedy dokonywana jest kompilacja.
4. Zmienna __cplusplus zdefiniowana jest tylko wtedy, gdy kompilowany jest program w języku C++.

Zmienne preprocesora

Wiele z tego zależy od implementacji, ale zazwyczaj zdefiniowane są następujące zmienne:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

1. __LINE__ i __FILE__ oznaczają, odpowiednio, numer linii źródłowej kompilowanego pliku i jego nazwę.
2. Zmienna __DATE__ zawiera datę — precyzyjniej datę kiedy dokonywana jest kompilacja.
3. Zmienna __TIME__ zawiera czas — precyzyjniej czas kiedy dokonywana jest kompilacja.
4. Zmienna __cplusplus zdefiniowana jest tylko wtedy, gdy kompilowany jest program w języku C++.
5. Zmienna __STDC__ zdefiniowana jest wtedy, gdy kompilowany jest program w języku C.

Inne przydatne stałe

- _WIN32 — Windows, wersja 32 bity,
- _WIN64 — Windows, wersja 64 bity,
- __APPLE__ — Apple,
- __linux__ — Linux,
- __unix__ — inny Unix,
- ...

Pełniejsza lista definicji akceptowanych przez większość kompilatorów: https://sourceforge.net/p/predef/wiki/OperatingSystems/ a lista definicji rozpoznawanych przez kompilator gcc tu: http://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html.

Makrorozwinięcia

Przykład

```
#ifdef DEBUG
# include <stdio.h>
# define D(x) do { printf("%s:%d (%s) ",\
                   __FILE__, __LINE__,\
                   __FUNCTION__); \
                   printf x ;\
                   fputc('\n', stdout); \
                   fflush(stdout); } \
              while (0);
#else
# define D(x)
#endif
```

Użycie makra D

```
D(("z=%d ", z))
```

Czemu tak?

Kompilacja warunkowa I

Preprocesor został wyposażony w kilka poleceń pozwalających na sterowanie przetwarzaniem kodu źródłowego. Są to:

```
#if warunek
/* ...          */
#endif

#if warunek
/* ...          */
#else
/* ...          */
#endif
```

Kompilacja warunkowa II

```
#if warunek1
/* ...          */
#elif warunek2
/* ...          */
#elif warunek3
/* ...          */
#else
/* ...          */
#endif
```

Dodatkowo sprawdzać można czy **zostały** zdefiniowane makra poleceniami

```
#ifdef makro
/* ...          */
#endif
```

Kompilacja warunkowa III

Oraz czy makro **nie zostało** zdefiniowane

```
#ifndef makro
/*  ...                 */
#endif
```

Kompilacja warunkowa

Po co?

1. Warunkowe włączanie kodu na potrzeby uruchomienia (DEBUG).
2. Tworzenie kodu przenośnego — w zależności od wersji systemu (kompilatora) standardowy zestaw makrodefinicji zawarty może być w plikach nagłówkowych.
3. „Zakomentowanie" dużego fragmentu kodu.