PREPROCESOR

Preprocesor jest wywoływany w pierwszym kroku tłumaczenia programu; przed rozpoczęciem właściwej kompilacji.

Preprocesor przetwarza tekst źródłowy progarmu !!!

Wstawianie plików

Często na początku programu pojawiają się jeden lub dwa wiersze postaci:

```
#include <nazwa_pliku>
albo
#include "nazwa_pliku"
```

Oznaczają one, że w miejscu wystąpienia polecenia **include** zostanie wstawiona zawartość pliku określonego przez `nazwa_pliku`.
`<nazwa_pliku>` oznacza, ze plik bedzie poszukiwany zgodnie z zasadami obowiązującymi w implementacji
`"nazwa_pliku"` oznacza, że poszukiwanie pliku zaczyna się tam, gdzie znaleziono progarm źródłowy, a jeśli w tym miejscu go nie ma, to plik jest poszukiwany zgodnie z zasadami obowiązującymi w implementacji.

Ograniczenia w przypadku użycia pierwszej formy:

- W `nazwa_pliku` nie może wystąpić znak `>` oraz znak nowego wiersza
- Skutek tej instrukcji jest nieokreślony, jeżeli wystąpi kt6óryś ze znaków `"` `'` `\` lub para znaków `/*` .

Ograniczenia w przypadku użycia drugiej formy:

- skutek użycia znaków `'` `\` oraz pary znaków `/*` jest nadal nieokreślony
- znak `>` jest dozwolony

Możliwe jest również użycie wiersza postaci:

```
#include ciag_leksemow
```

Ciąg leksemów jest rozwijany wówczas zgodnie ze zwykłymi regułami, natomiast w wyniku musi doprowadzić do jednej z postaci: `<...>` albo `"..."`.

Stosowanie `#include` jest zalecanym sposobem sporządzania deklaracji dla dużego programu. Gwarantuje to identyczność *definicji i deklaracji* zmiennych i funkcji.
Uwaga: Wączanie plików może być zagnieżdżone.

Makrorozwinięcia – zastąpienie nazwy przez ciąg znaków

Zasęgo dyrektywy – od miejsca wystąpienia do końca pliku.

Wiersz sterujący o postaci

```
#define identyfikator  ciag-leksemow
```

zleca preprocesorowi zastępowanie dalszych wystąpień identyfikatora wskazanym ciągiem leksemów. Opuszca się odstępy otaczające ciąg leksemów.
Uwaga Ponowne wystąpienie `#define` z tym samym identyfikatorem traktuje się jako błędne, jeżeli ciągi leksemów nie są w obu przypadkach identyczne (przy tym wszystkie białe plamy rozdzielające leksemy traktuje się jako równoważne).

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

Następnie można pisać:

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

Makro z argumentami

```
#define identyfikator(lista-identyfikatorow)   \
                                    ciag-leksemow
```

Przykład:

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

Nawiasy w definicji sa istotne, ponieważ np.

```
x=max(p+q,r+s);
                zostanie zastapione przez
x=((p+q)>(r+s)?(p+q):(r+s);
```

Kilka reguł:

- definicja może korzystać z poprzednich definicji (zagnieżdżenie),
- makrorozwinięcie nie obowiązuje w stałych tekstowych ograniczonych znakami cudzysłowu, np.

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

  nie spowoduje zastąpienia YES.

Przykład 2:

```
/*  makro zamieniajace miejscami argumenty */
#define SWAP(x,y)   {                          \
                      double tmp;              \
                      tmp=x;                   \
                      x=y;                     \
                      y=tmp;                   \
                    }
```

Przykład 3:

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

UWAGA: Należy unikać jako argumentów wyrażeń mających efekty uboczne, a także wywołań funkcji, ponieważ po rozwinięciu makrodefinicji tekst może zostać wstawiony kilkakrotnie.

Przykłady:

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

maximum=max(++a,10);
    /* wynikiem b\c edzie 10 albo  a+2  */
    /* a bedzie zwiekszone o 1 lub 2    */

maximum=max(fgetc(file),maximum);
    /* tak mozna sasiada najwiekszego
       znaku w pliku                    */

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

Uwaga: Ostrożnie z używaniem średnika w makrodefinicjach.

Wiersz sterujący o postaci

```
#undef identyfikator
```

zleca preprocesorowi, aby zapomniał definicję identyfikatora. Zastosowanie `#undef` do nie zdefiniowanego identyfikatora nie jest błędem.

Znaki `#` i `##`

Jeśli parametr poprzedzany znakiem `#`, to identyfikator parametru wraz ze znakiem `#` zostaną zastąpione odpowiednim argumentem otoczonym znakami cudzysłowu `"`.

Każdy ze znaków `"` i `\` występujący na początku, w środku lub na końcu stałych znakowych i napisów tworzących argument zostanie poprzedzony znakiem `\`.

Jeśli w ciągu leksemów definiujących makro wystąpi operator `##`, to po zastąpieniu parametrów napisami – operator `##` wraz z otaczającymi go białymi plamami zostanie usunięty.

Jeśli tak utworzony leksem jest niepoprawny lub wykonanie zależy od kolejności, to skutek jest niezdefiniowany.

Niezależnie od postaci makrodefinicji zastępujący ciąg leksemów jest wielokrotnie przeglądany w poszukiwaniu innych tak zdefiniowanych identyfikatorów. Jeżeli jednak identyfikator zastąpiony już w danym rozwinięciu, znów pojawi się przy ponownym przeglądaniu, to pozostanie w rozwiniętym tekście bez zmiany.

Mechanizm makrodefinicji przydaje się do definiowania "wżnych stałych

Przykład:

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

Definiuje makro zwracające wartość absolutną różnicy jego argumentów.
W przeciwieństwie do funkcji zwracana wartość może być dowolnego typu.

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

Wywołanie `tempfile(/usr/tmp)` daje w wyniku

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

co następnie zostaje sklejone w jeden napis. Przy definicji

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

Wywolanie

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

produkuje leksem

```
var123
```

Wynik wywolania

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

jest nieokreślony; obecność operatora `##` zapobiega rozwinięciu argumentów wewnętrznego wywołania.

W rezultacie powstanie `cat( 1 ,2 )3`, w którym leksem `)3` (wynik sklejenia ostatniego leksemu pierwszego argumentu z pierwszym leksemem drugiego jest niepoprawny.

Po wprowadzeniu makra drugiego poziomu

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

wszystko to działa bardziej przyzwoicie: *xcat(xcat(1,2),3)* rzeczywiście tworzy leksem *123*, ponieważ w samym xcat nie występuje operator `##`.

Podobnie `ABSDIFF(ABSDIFF(a,b),c)` tworzy wynik w pełni rozwinięty zgodnie z oczekiwaniem.

Kompilacja warunkowa

Fragmenty programu mogą być kompilowane warunkowo zgodnie z poniższą schematyczną składnią:

```
Kompilacja warunkowa:
     wiersz-if tekst czesci-elif czesc-else  #endif

     wiersz_if:
        #if wyrazenie-stale
        #ifdef identyfikator
        #ifndef identyfikator

     czesci el-if:
        wiersz-elif tekst
        czesci-elif

     wiersz-elif:
        #elif wyrazenie stale

     czesc-else
        wiersz_else tekst

     czesc-else:
        wiersz-else tekst

     wiersz-else:
        #else
```

Kilka reguł:

- Wyrażenia stałe występujące w `#if` i `#elif` są obliczane kolejno, aż do wystąpienia wyrażenia z niezerową wartością.
- Tekst następujący po wyrażeniach z zerową wartością opuszcza się w czasie kompilacji.
- Tekst występujący po wyrażeniach z pozytywną wartością jest włączany do programu.
- Po znalezieniu wyrażenia z pozytywną wartością i jego obsłużeniu to kolejne wiersze `#elif` lub `#else` są pomijane wraz z ich tekstami.
- Jeżeli wszystkie wyrażenia są równe zero i występuje `#else`, to obsługuje się tekst następujący po `#else`.
- Tekst występujący w gałęziach nieaktywnych jest ignorowany, ale sprawdza się najpierw, czy nie występują w nim zagnieżdżone konstrukcje warunkowe.

Wyrzżenia stałe w `#if` i `#elif` są przedmiotem zwykłych makrorozwinięć.

Wyrażenia postaci:

```
    defined identyfikator
lub
    defined ( identyfikator )
```

przed rozwijanie makr otrzymują wartość:

1L - jeśli identyfikator jest zdefiniowany w preprocesorze

0L - jeśli nie jest zdefiniowany

Wszystkie identyfikatory pozostałe po makrorozwinięciach otrzymują wartość `0L` , a cała arytmetyka na stałych jest przeprowadzana na liczbach całkowitych długich lub długich bez znaku.

Wynikowe wyrażenie stałe ma ograniczenia:

- musi być całkowite
- nie może zawierać operatora *sizeof*, rzutowania i stałych wyliczeń.

Wiersze sterujące postaci:

```
#ifdef identyfikator
#indef identyfikator
```

są równoważne:

```
#if defined identyfikator
#if ! defined identyfikator
```

Przykłady:

```
1)  /* Kompilowanie programu w kilku wersjach  */

#define DEMO
#ifdef DEMO
/*  kod tylko dla wersji demonstracyjnej  */

 instrukcje

#endif

#ifndef DEMO
/*  Kod tylko dla wersji normalnej  */
#endif

2)  /*  Uniemozliwienie wielokrotnego przetwarzania
        tego samego kodu zrodlowego  */

    /*  poczatek pliku naglowkowego  okienka.h  */
    #ifndef OKIENKA
    #define OKIENKA

    /*  tresc pliku  */
    #endif
    /*  Koniec pliku okienka  */
```

Numeracja wierszy

Dla potrzeb innych preprocesorów, które generują programy w języku C, wiersz mający jedną z postaci

```
#line stala "nazwa-pliku"
#line stala
```

zleca kompilatorowi, aby dla celów diagnostycznych przyjął, że następny wiersz źródłowy będzie miał numer określony przez stałą, a nazwą bieżącego pliku źródłowego będzie *nazwa-pliku*.

Makra występujące w takich wierszach są rozwijane przed interpretacją instrukcji.

Generowanie błędów

Wiersz sterujący o postaci

```
#error ciag-leksemow
```

zleca procesorowi wypisanie komunikatu diagnostycznego zawierającego podany ciąg leksemów.

Instrukcja *pragma*

Wiersz sterujący o postaci

```
#pragma ciag-leksemow
```

zleca procesorowi podjęcie akcji zależnej od implementacji. Nieznana akcja jest ignorowana.

Pusta instrukcja preprocesora

Wiersz zawierający jedynie znak `#` nie ma żadnego skutku.

Nazwy zdefiniowane w preprocesorze

Definicji tych nazw, jak rówmież operatora **defined** (występującego w wyrażeniach preprocesora), nie można odwołać ani zmienić.

- __LINE__ – Dziesiętna stała całkowita zawierająca numer bieżącego wiersza programu źródłowego
- __FILE__ – Stała napisowa zawierająca nazwę tłumaczonego pliku
- __DATE__ – Stała napisowa zawierająca datę tłumaczenia programu w formacie – `"Mmm dd rrrr"`
- __TIME__ – Stała napisowa zawierajęca czas tłumaczenia programu w formacie – `"gg:mm:ss"`
- __STDC__ – Stała *1*. Z zamierzenia identyfikator ten powinien być zdefiniowany z wartością *1* jedynie w implementacjach dostosowanych do standardu.