### Загальні відомості

В інтегровану середу підготовки програм на Сі або в компілятор мови як обов'язковий компонент входить препроцесор. Призначення препроцесора - обробка вихідного тексту програми до її компіляції. Препроцесорна обробка включає кілька стадій, які виконуються послідовно. Конкретна реалізація може об'єднувати кілька стадій, але результат повинен бути таким, як якби вони виконувалися в наступному порядку:

1. Всі системно-залежні позначення перекодуються в стандартні коди.
2. Кожна пара з символів '\' і "кінець рядка" разом з пробілами між ними прибираються, і тим самим наступний рядок вихідного тексту приєднується до рядка, в якій знаходилася ця пара символів.
3. У тексті розпізнаються директиви і лексеми препроцесора, а кожен коментар замінюється одним символом порожнього проміжку.
4. Виконуються директиви препроцесора і виробляються макропідстановки.
5. *Ескейп-послідовності* в символьних константах і символьних рядках замінюються на їх еквіваленти.
6. Суміжні символьні рядки конкатенуються, тобто з'єднуються в один рядок.
7. Кожна препроцесорна лексема перетворюється в текст на мові Сі.

Пояснимо, що розуміється під препроцесорною лексемамою або лексемами препроцесора. До них відносяться символьні константи, імена включених файлів, ідентифікатори, знаки операцій, препроцесорні числа, знаки пунктуації, рядкові константи і будь-які символи, відмінні від пробілу.

Стадія обробки директив препроцесора. При її виконанні можливі наступні дії:

* заміна ідентифікаторів заздалегідь підготовленими послідовностями символів;
* включення в програму текстів із зазначених файлів;
* виключення з програми окремих частин її тексту, умовна компіляція;
* макропідстановка, тобто заміна позначення параметризованим текстом, який формується препроцесором з урахуванням конкретних аргументів.

### Символьні константи: #define

Якщо в якості першого символу в рядку програми використовується символ #, то цей рядок є командним рядком препроцесора (макропроцесора). *Командний рядок* препроцесора закінчується символом переведення на новий рядок. Якщо безпосередньо перед кінцем рядка поставити символ зворотної косої межі "\", то командний рядок буде продовжений на наступний рядок програми.

*Директива #define,* подібно всім директивам препроцесора, починається з символу # в самій лівій позиції. Вона може з'явитися в будь-якому місці вихідного файлу, а дається визначення має силу від місця появи до кінця файлу. Ми активно використовуємо цю директиву для визначення символічних констант в наших прикладах програм, однак вона має більш широке застосування, що ми покажемо далі.

#### Заміна ідентифікаторів

#define ідентифікатор рядок

Приклад:

#define ABC 100

Замінює кожне входження ідентифікатора ABC в тексті програми на 100:

#undef ідентифікатор

Приклад:

#undef ABC

Скасовує попереднє визначення для ідентифікатора ABC.

Приклад:

/\* Прості приклади директиви препроцесора \*/

#define TWO 2 /\* можна використовувати коментарі \*/

#define MSG "Текст 1.\

Продовження тексту 1"

/\* зворотна коса риска продовжує визначення на наступний рядок \*/

#define FOUR TWO\*TWO

#define PX printf("X дорівнює %d.\n", x)

#define FMT "X дорівнює %d.\n"

int main( )

{

int x = TWO;

PX;

x = FOUR;

printf(FMT,x);

printf("%s\n",MSG);

printf("TWO: MSG\n");

return TWO;

}

В результаті виконання нашого прикладу матимемо:

X дорівнює 2

X дорівнює 4

Текст 1. Продовження тексту 1

TWO: MSG

Розберемо, що сталося. Оператор

int x = TWO;

перетворюється в

int x = 2;

Потім оператор

PX;

Перетворюється в

printf("X дорівнює %d.\n",x);

оскільки зроблена повна заміна. Тепер ми бачимо, що макроозначення може представляти будь-який рядок, навіть цілий вираз на мові Сі. Зауважимо, що це константний рядок. PX надрукує тільки змінну, названу x.

У наступному рядку виконується наступне:

x = FOUR;

перетворюється

x = TWO\*TWO;

перетворюється в

x = 2\*2;

і на цьому все закінчується. Фактичне множення має місце не під час роботи препроцесора і не при компіляції, а завжди без винятку при роботі програми (Уточнення: це залежить від конкретного компілятора). **Препроцесор не виконує обчислень**. Він тільки дуже точно робить запропоновані підстановки. Зауважимо, що макроозначення може включати інші визначення. Деякі компілятори не підтримують цю властивість вкладення. У наступному рядку

printf(FMT,x);

перетворюється в

printf("X дорівнює %d.\n",x)

коли FMT замінюється відповідним рядком. Цей підхід може виявитися дуже зручним, якщо є довгий рядок, який ми використовуємо кілька разів. У наступному рядку програми MSG замінюється відповідним рядком. Лапки роблять замість попереднього рядоку константою символьного рядка. Оскільки програма отримує її вміст, цей рядок буде запам'ятовуватися в масиві, що закінчується нуль-символом. так,

#define HAL 'X' визначає символьну константу, а

#define HAR "X" визначає символьний рядок X\0

Зазвичай препроцесор, зустрічаючи одне з макроозначень в програмі, дуже точно замінює їх еквівалентної рядком заміщення. Якщо цей рядок також містить макроозначення, вони теж заміщаються. Єдиним винятком при заміні є макроозначень, що знаходиться всередині подвійних лапок. Тому

printf("TWO: MSG");

друкує буквально TWO: MSG замість друку наступного виду:

2: "Текст 1.

Продовження тексту 1"

Якщо нам потрібно надрукувати цей текст, можна використовувати оператор

printf("%d: %s\n",TWO,MSG);

тому що тут макроозначення знаходяться поза лапок.

Коли слід використовувати символічні константи? Ймовірно, ми повинні застосовувати їх для більшості чисел. Якщо число є константою, використовуваною в обчисленнях, то символічне ім'я робить ясніше її сенс. Якщо число - розмір масиву, то символічне ім'я спрощує зміна вашої програми при роботі з великим масивом. Якщо число є системним кодом, скажімо для символа EOF, то символічне уявлення робить програму більш переносимою. Змінюється тільки визначення EOF. Мнемонічне значення, легкість зміни, переносимість: все це робить символічні константи заслуговують на увагу!

### Використання аргументів з #define

Щоб уникнути помилок при обчисленні виразів параметри макроозначення необхідно укладати в дужки.

#define ідентифікатор 1 (ідентифікатор2, . . .) рядок

Приклад:

#define abs(A) (((A) > 0)?(A) : -(A))

Кожне входження виразу abs(arg) в тексті програми замінюється на

((arg) > 0) ? (arg) : -(arg),

причому параметр макроозначення А замінюється на arg.

Приклад:

#define nmem(P,N)\

(P) -> p\_mem[N].u\_long

Символ \ продовжує макроозначення на другий рядок. Це макроозначення зменшує складність вираження, що описує масив об'єднань усередині структури.

Макроозначення з аргументами дуже схоже на функцію, оскільки аргументи його укладені в дужки:

/\* макровизначення з аргументами \*/

#define SQUARE(x) x\*x

#define PR(x) printf("x дорівнює %d.\n", x)

int main( )

{

int x = 4; int z;

z = SQUARE(x);

PR(z);

z = SQUARE(2);

PR(z);

PR(SQUARE(x));

PR(SQUARE(x+2));

PR(100/SQUARE(2));

PR(SQUARE(++x));

return 0;

}

Усюди, де в нашій програмі з'являється макроозначення SQUARE (x), воно замінюється на x \* x. На відміну від наших колишніх прикладів, при використанні цього макроозначення ми можемо абсолютно вільно застосовувати символи, відмінні від x. У макровизначенні 'x' заміщується символом, використаним в макровизові програми. Тому макроозначенняSQUARE (2) заміщується на 2 \* 2. Таким чином, x діє як аргумент. Однак, аргумент макроозначення не працює - точно так само, як аргумент функції. Ось результати виконання програми:

z дорівнює 16.

z дорівнює 4.

SQUARE(x) дорівнює 16.

SQUARE(x+2) дорівнює 14.

100/SQUARE(2) дорівнює 100.

SQUARE(++x) дорівнює 36.

Перші два рядки очевидні. Зауважимо, що навіть всередині подвійних лапок в визначенні PR змінна заміщується відповідним аргументом. Всі аргументи в цьому визначенні заміщаються. Розглянемо третій рядок:

PR(SQUARE(x));

Вона стає наступним рядком:

printf("SQUARE(x) дорівнює %d.\n", SQUARE(x));

після першого етапу макророзширення. Друге SQUARE (x) розширюється, перетворюючись на x \* x, а перше залишається без зміни, тому що тепер воно знаходиться всередині лапок в операторі програми, і таким чином захищене від подальшого розширення. Остаточно рядок програми містить

printf("SQUARE(x) дорівнює %d.\n",x\*x);

і виводить на друк

SQUARE(x) дорівнює x\*x.

Якщо макроозначення включає аргумент з подвійними лапками, то аргумент буде заміщатися рядком з макровиклику. Але після цього він надалі не розширюється, навіть якщо рядок є ще одним макроозначенням. У нашому прикладі змінна x стала макроозначення SQUARE (x) і залишилася ним. Згадаймо, що x = 4. Це дозволяє припустити, що SQUARE (x + 2) буде дорівнює 6 \* 6 або 36. Але надрукований результат говорить, що виходить число 14. Причина такого результату така: препроцесор не робить обчислень. Він тільки заміщає рядок. Усюди, де наше визначення вказує на x, препроцесор підставить рядок x + 2.

Таким чином,

x\*x становится x+2\*x+2

Якщо x дорівнює 4, то виходить

4+2\*4+2=4+8+2=14

Виклик функції передає значення аргументу в функцію під час виконання програми. Макровиклик передає рядок аргументів на програму до її компіляції.

### Макроозначення або функція?

|  |  |
| --- | --- |
| ! | Багато задач можна вирішувати, використовуючи макровизначення з аргументами або функцію. Що з них слід застосовувати? На цей рахунок немає строгих правил, але є деякі міркування. |

*Макроозначення* повинні використовуватися скоріше як хитрість, а не як звичайні функції. Вони можуть мати небажані побічні ефекти. Деякі компілятори обмежують макроозначення одним рядком, і, мабуть, краще дотримуватися такого обмеження, навіть якщо ваш компілятор цього не робить.

Вибір макроозначення призводить до збільшення обсягу пам'яті, а вибір функції - до збільшення часу роботи програми. Макроозначення створює рядковий код, тобто ми отримуємо оператор в програмі. Якщо макроозначень застосувати 20 разів, то в програму вставиться 20 рядків коду. Якщо ми використовуємо функцію 20 разів, то у нас буде тільки одна копія операторів функції. Однак управління програмою слід передати туди, де знаходиться функція, а потім повернутися в зухвалу програму, а на це потрібно більше часу, ніж при роботі з малими кодами. Так що думайте, що вибирати!

Перевага макроозначення полягає в тому, що при їх використанні нам не потрібно турбуватися про типи змінних, тому що макроозначення мають справу з символьними рядками, а не з фактичними значеннями. Tак наше макроозначення SQUARE (x) можна використовувати однаково добре зі змінними типу int або float.

**Запам’ятаємо**!

1. У макровизначеннях немає прогалин, але вони можуть з'явитися в замісному рядку. Препроцесор вважає, що макроозначення закінчується на першій прогалині, тому все, що стоїть після пробілу, залишається в замісному рядку.
2. Використовуйте круглі дужки для кожного аргумента і всього визначення. Це є гарантією того, що елементи будуть згруповані належним чином в вираженні.
3. Для імен макрофункцій слід використовувати великі літери. Ця угода не поширюється так широко, як угода про використання великих літер для макроконстант. Їх застосування застереже від можливих побічних ефектів макроозначень.

Припустимо, що ми розробили кілька макрофункцій на свій розсуд. Якщо ми пишемо нову програму, ми не повинні їх перевизначати. Потрібно використовувати директиву #include.

### 

### Включення файлу: #include

Перелік позначень заголовків файлів для роботи з бібліотеками компіляторів затверджений стандартом мови. Нижче наведені назви цих файлів, а також короткі відомості про ті описи і визначення, які в них включені. Більшість описів - прототипи стандартних функцій, а визначені в основному константи, наприклад EOF, необходимые для работы с библиотечными функциями.

assert.h – *діагностика програм*

ctype.h – перетворення та перевірка символів

errno.h – перевірка помилок

float.h – робота з речовими даними

limits.h - граничні значення цілочисельних даних

locale.h - підтримка національного середовища

math.h – математичні обчислення

setjump.h – можливості нелокальних переходів

signal.h – обробка виключних ситуацій

stdarg.h - підтримка змінного числа параметрів

stddef.h – додаткові визначення

stdio.h - засоби введення-виведення

stdlib.h - функції загального призначення (робота з пам'яттю)

string.h - робота з рядками символів

time.h – визначення дати та часу

У конкретних реалізаціях кількість і найменування заголовних файлів можуть бути і іншими. Наприклад, в компіляторах для MS-DOS активно використовуються заголовки mem.h, alloc.h, conio.h, dos.h та інші. У компіляторах Turbo C, Borland C ++ для зв'язку з графічною бібліотекою застосовується заголовочний файл graphics.h.

*Командний рядок*  #include може зустрічатися в будь-якому місці програми, але зазвичай всі включення розміщуються на початку файлу вихідного тексту.

#include <ім’я\_файлу>

Приклад:

#include <math.h>

*Препроцесор заміняє цей рядок вмістом файла* math.h. Кутові дужки означають, що файл math.h буде взятий з деякого стандартного каталогу (зазвичай це /usr/include ). *Поточний каталог не проглядається*:

#include " ім’я\_файлу "

Приклад:

#include "ABC"

Препроцесор заміняє цей рядок вмістом файла ABC. Так як ім'я файлу укладено в лапки, то пошук проводиться в поточному каталозі (в якому міститься основний файл вихідного тексту). Якщо в поточному каталозі даного файлу немає, то пошук проводиться в каталогах, визначених ім'ям шляху в опції -l препроцесора. Якщо і там немає файлу, то проглядається стандартний каталог.

В операційній системі UNIX кутові дужки повідомляють препроцесору, що файл слід шукати в одному або декількох стандартних системних каталогах. Лапки кажуть йому, що спочатку потрібно дивитися в вашому каталозі або в якомусь іншому, якщо ви визначаєте його ім'ям файлу, а потім шукати в стандартних місцях.

У конкретних реалізаціях кількість і найменування заголовних файлів можуть бути різними:

#include <stdio.h> шукає в системному каталозі

#include "my.h" шукає в поточному робочому

каталозі

#include "/user/1/my.h" шукає в каталозі /user/1

У типовій мікропроцесорної системі ці дві форми є синонімами, і препроцесор веде пошук на вказаному диску.

#include "stdio.h" шукає на стандартному диску

#include <stdio.h> шукає на стандартному диску

#include "a:stdio.h" шукає на диску а

За угодою суфікс .h використовується для заголовків файлів, тобто файлів з інформацією, яка розташовується на початку програми. Заголовки зазвичай складаються з операторів препроцесора.  
Деякі файли включені в систему, наприклад, stdio.h, але можна створити і свій файл.  
Багато програмістів розробляють свої стандартні заголовки, щоб використовувати їх в програмах.

### Умовна компіляція

Командні рядки препроцесора використовуються для умовної компіляції різних частин вихідного тексту в залежності від зовнішніх умов.

#if константний\_вираз

Приклад:

#if ABC + 3

*Істина,* якщо константний виразABC + 3 не дорівнює нулю.

#ifdef ідентифікатор

Приклад:

#ifdef ABC

*істина,* якщо ідентифікатор *ABC* визначено раніше командою *#define*.

#ifndef ідентифікатор

Приклад:

#ifndef ABC

істина, якщо ідентифікатор ABC не визначений зараз.

#else

. . .

#endif

Якщо попередні перевірки #if, #ifdef або #ifndef дають значення "Істина", то рядки від #else до #endif ігноруються при компіляції.

Якщо ці перевірки дають значення "Брехня", то рядки від перевірки до #else (а при відсутності #else - до #endif) ігноруються.

Команда #endif позначає кінець умовної компіляції.

Приклад:

#ifdef DEBUG

fprintf (stderr, "location: x = %d\n", x);

#endif

#### **допоміжні директиви**

*Номер рядка і ім'я файлу*

#line ціла\_константа "ім’я\_файлу"

Приклад:

#line 20 "ABC"

Препроцесор змінює номер поточного рядка і ім'я компилируемого файлу. Файл може бути опущено.  
Одна з цілей використання умовної компіляції - зробити програму більш мобільною. Змінюючи кілька ключових визначень на початку файлу, ми можемо встановлювати різні значення і включати різні файли для різних систем.

Приклад:

#define N 3/\*визначення константи \*/

void main( )

{

#line 55 "file.c"

double x[3\*N];

}

#### Реакція на помилки

#error послідовність лексем

Обробка директиви призводить до видачі діагностичного повідомлення у вигляді, визначеному послідовністю лексем. Застосування цієї директиви спільно з умовними препроцесорну командами.

Приклад:

#define NAME 15

Надалі можна перевірити її значення і видати повідомлення, якщо у NAME виявиться інше значення:

#if (NAME !=15)

#error NAME повинне дорівнювати 15!

Повідомлення буде виглядати так:

error <ім’я\_файлу><номер\_рядка >;

error directive: NAME має дорівнювати 15!

#### Пуста директива

#

Використання цієї директиви не викликає ніяких дій.

#### Прагми

#pragma

Ця директива визначає дії, що залежать від конкретної реалізації компілятора. Наприклад в деякі компілятори входить варіант цієї директиви для сповіщення компілятора про наявність в тексті програми команд на мові Асемблер. Можливості команди #pragma можуть бути разнообазнимі. Стандарту для них не існує. Якщо конкретний препроцесор зустрічає ПрагмУ, яка йому невідома, він її просто ігнорує як порожню директиву. У деяких реалізаціях включена прагма.

#pragma pack(n), где n= 1, 2, 4. Прагма pack дозволяє впливати на упаковку суміжних елементів в структурах і об'єднаннях (див. Лекцію 14).

Угода може бути такою:

pack(1) - вирівнювання елементів по межах байтів;

pack(2) - вирівнювання елементів по кордонах слів;

pack(4) - вирівнювання елементів по кордонах подвійних слів;

В деякі компілятори включені Прагма, що дозволяють змінювати спосіб передачі параметрів функцій, порядок приміщення параметрів в стек і т.д.

**Вбудовані макроімена**

Існують вбудовані (заздалегідь певні) макроімена, доступні препроцесору під час обробки. Вони дозволяють отримати наступну інформацію:

\_\_DATE\_\_ - рядок символів в форматі: "місяць число рік", яка визначає дату початку обробки вихідного файлу. Наприклад, після препроцесорної обробки тексту програми, виконаної 29 січня 2005 року, оператор

printf(\_\_DATE\_\_);

стане таким

printf("%s", "January 29 2005");

\_\_LINE\_\_ - десяткова константа - номер поточного оброблюваного рядка файлу з програмою на Сі. Прийнято, що номер першого рядка вихідного файлу дорівнює 1 ;

\_\_FILE\_\_ - рядок символів - ім'я компілюємого файлу. Ім'я змінюється щоразу, коли препроцесор зустрічає директиву #include із зазначенням імені іншого файлу. Коли включення файлу по команді #include завершуються, встановлюватися попереднє значення макроімені\_\_FILE\_\_ ;

\_\_TIME\_\_ - рядок символів виду "годинник: хвилини: секунди", яка визначає час початку обробки препроцесором вихідного файлу;

\_\_STDC\_\_ - константа, рівна 1, якщо компілятор працює відповідно до ANSI-стандартом. В іншому випадку значення мікроімені \_\_STDC\_\_ не визначене. Стандарт мови Сі передбачає, що наявність імені \_\_STDC\_\_ визначається реалізацією, так як макрос \_\_STDC\_\_ ставиться до нововведень стандарту. У конкретних реалізаціях набір визначених імен набагато ширше. Для отримання більш повних відомостей про зумовлених препроцесорну іменах слід звертатися до документації по конкретному компілятору.