Макровизначення

Макровизначення (макропідстановка або просто макрос) в мові С(С++) – це вираз, який при компіляції файлу з кодом програми підставляється замість імені, що визначає дану макропідстановку. Макровизначення обробляються ***препроцесором*** – програмою, яка аналізує програмний код і готує його для роботи компілятора. Команди, які має обробляти препроцесор, записуються у вигляді директив. Директиви препроцесора починаються із знака **#**:

```
#define <ідентифікатор>   <текст макросу>
```

Приклади:

```
#define MAX_SIZE 10
int main ()
{
  int arr [MAX_SIZE]; // MAX_SIZE заміняється на 10
  // інші інструкції
}
```

Зауваження

1. Зверніть увагу – текст макросу просто “підставляється”, тому макровизначення виду:

   ```
   #define MAX_SIZE = 10
   ```

   або

   ```
   #define MAX_SIZE 10;
   ```

   є помилковими.
2. Допускається визначення макросів з параметрами – звертання до них виглядає як звертання до функцій. Багато стандартних функцій бібліотек C/C++ є макровизначеннями.
3. Директива `#undef <ідентифікатор>` зупиняє дію ідентифікатора, який був визначений раніше і дозволяє його повторне визначення в інших цілях.
4. Операція `##` препроцесором сприймається як операція конкатенації (тобто зчеплення або з'єднання).

Директива макровизначення з параметрами:

```
#define <ідентифікатор>(<параметри>)<текст макросу>
```

Дуже важливе зауваження:

Між ідентифікатором макросу та круглою дужкою, що відкриває список його параметрів, пробілу немає.

Приклад:

```
#define cube(x) x*x*x
int main ()
{
 int i = 1;
 float x = 1.0;
 cout << cube (i) << endl;// універсально для всіх
 cout << cube (x) << endl;// типів параметрів!
}
#undef cube
```

Ще один приклад:

```
// Цей знак ## конкатенує аргументи
#define concat(left, right) left ## right
int main ()
{
// визначаємо складений ідентифікатор  ab
 int concat (a, b) = 100;
// використовуємо цей ідентифікатор
 cout << concat (a, b) << endl;
}
```

Ще один приклад:

```
#define max(x, y)  (x > y)  ? x : y
int main ()
{
 int i = 1, j = 2;
 float x = 1.0, y = 2.0;
 int m = max(i, j);
 cout << m << endl;
 float f  = max(x, y);
 cout << f << endl;
}
```

Проте – спробуємо проаналізувати, як спрацюють дані макровизначення у наступних викликах:

```
int main ()
{
 int i = 10, j = 20;
 float x = 10.0, y = 20.0;
 cout << cube (i + j) << endl; //???
 cout << max (x, y) << endl;   //???
}
```

Останній рядок взагалі приведе до синтаксичної помилки – низький пріоритет умовного виразу диктуватиме першочергове виконання потокових операцій << :

```
(cout << (x > y)) ? x : (y << endl);
```

Внесемо очевидні виправлення у тексти макровизначень:

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

Тепер їх використання не приводить до проблем:

```
int main ()
{
 int i = 1, j = 2;
 float x = 1.0, y = 2.0;
 cout << cube (i + j) << endl; //!!!
 cout << max (x, y) << endl;   //!!!
}
```

Проте – чи правильно ви визначите результати звертань:

```
cout << cube (i++) << endl; //???
cout << cube (++i) << endl; //???
```

Директиви компіляції

Ще одна група директив препроцесора пов'язана з компіляцією програмних файлів. Для підключення так званих ***header-файлів*** (файлів заголовків), тобто файлів, які містять визначення констант та макросів, декларації функцій, класів, шаблонів, тощо, використовуються директиви

```
#include <ідентифікатор header-файлу>
```

та

```
#include “ідентифікатор header-файлу”
```

У першому випадку відповідний файл має знаходитись у системних директоріях, а в другому – у поточній директорії, яка і містить файл, що компілюється.

Інша група директив використовується для так званої умовної компіляції – керування роботою препроцесора.

Це директиви **#if**, **#elif**, **#else** та **#endif**. Порядок їх використання наступний :

```
#if константний_вираз_1
// тут деякі директиви
   #elif константний_вираз_2
// тут деякі директиви
   #else
// тут деякі директиви
#endif
```

Якщо цілий константний вираз у директиві **#if** має ненульове значення (ІСТИНА), то при компіляції включаються всі наступні рядки до **#elif** або **#endif** або **#else** (**elif** діє як гілка **else-if**).

Приклад.

Змінюючи константу VERSION, керуємо включенням файлів:

```
#define VERSION 3
#if VERSION == 1
      #define INCLUDE_FILE "file_1.h"
   #elif VERSION == 2
      #define INCLUDE_FILE "file_2.h"
   #else
      #define INCLUDE_FILE "file_3.h"
#endif
#include INCLUDE_FILE
```

Для того, щоб позбавитись повторних включень файлів заголовків, використовують наступні директиви:

```
#ifndef <ідентифікатор>
#define <ідентифікатор>

// код, який включається у компіляцію

#endif
```

Перша директива перевіряє, чи був визначений `<ідентифікатор>` директивою `#define` . Якщо ні, то наступна директива його визначає і при компіляції включається текст аж до директиви `#endif`.

Інструкція typedef

Інструкція **typedef** дозволяє визначати альтернативне ім'я для існуючого типу.

Приклад.

```
typedef unsigned short WORD_;
```

Тепер **WORD_** є новою назвою для типу **unsigned short**. Тому допустима декларація:

```
WORD_ val; // val має тип unsigned short
```

Приклад.

```
typedef struct //визначаємо структуру matrix
{
   float mas [NR][NC];
   int nr;   int nc;
} matrix;
```

Тепер **matrix** є назвою для структури визначеного вище типу.

З допомогою інструкції **typedef** можна визначити і тип вказівника на функцію:

```
typedef double (*p_function)
  (double);
```

Тепер **p_function** – це назва типу вказівника на функцію, яка приймає аргумент типу **double** та повертає результат типу **double**. Цьому вказівнику можна присвоїти ідентифікатор будь-якої функції відповідного виду (ідентифікатор функції є вказівником на неї) або передати у функцію в якості параметра.