1.11. Препроцессор

Monday, July 6, 2020 19:42

В этой главе мы рассмотрим еще одну фичу языка C, которой нет у многих других высокоуровневых языков программирования. Препроцессор предоставляет инструментарий для упрощения разработки, портируемости программ на другие системы, изменения кода и его чтения. Также можно использовать препроцессор, чтобы настроить язык C под себя и свои нужды. В этой главе мы рассмотрим:

- Создание констант и макросов при помощи `#define`
- Работу с библиотеками при помощи `#include`
- Директивы `#ifdef, #endif, #else и #ifndef`

Препроцессор анализирует относящиеся к нему директивы до того, как начнется анализ основного кода программы. Все директивы препроцессора начинаются с **хэша** (#), причем перед этим знаком ничего кроме пробелов быть не должно. Возможно, вы уже заметили, что в каждой программе мы использовали директиву препроцессора `#include`. В этом уроке мы разберем его подробнее, но сначала разберем директиву `#define`.

Директива #define

Одно из основных применений `#define` – присвоение имен константам, то есть создание **символьных констант**. Например, выражение

```
#define   YES    1
```

задает числу 1 соответствующее ему слово `YES`. Теперь слово `YES` можно использовать в программе как замену числу 1. Например, в выражении

```
gameOver = YES;
```

происходит присвоение значения `YES` переменной `gameOver`. Таким образом, можно легко прочитать и при необходимости изменить значение `YES`, а также при правильно выбранном названии легко понять, что хранится в переменной `YES`, – в данном случае имееся в виду логическое 1. Такую константу также можно использовать в выражениях:

```
if ( gameOver == YES )
      ...
```

Единственный случай, где константу использовать нельзя – это внутри строки, т.е. указатель

```
char  *charPtr = "YES";
```

будет указывать именно на строку со словом `"YES"`, а не на строку `"1"`.

Символьная константа – это **не переменная**. Операция присваивания здесь неприменима, потому что препроцессор при компиляции просто ищет в коде обращения к этому имени и заменяет его на соответствующее ему значение. Также обратите внимание, что при объявлении константы через `#define` не используется никаких дополнительных знаков, даже знака равенства и точки с запятой. Позже разберем, почему так. А пока напишем программу, иллюстрирующую работу с `#define`. Программа на вход будет получать значение радиуса и считать площадь круга, длину окружности и объем сферы с таким радиусом. Зададим константу PI, которая будет использоваться при вычислении вышеописанных значений.

прим.: в библиотеке `<math.h>` уже есть константа `M_PI`, которой можно пользоваться, если мы добавим эту библиотеку в программу через `#include`, но мы зададим свою константу.

```
1    /* Function to calculate the area and circumference of a
2       circle, and the volume of a sphere of a given radius  */
3
```

```c
#include <stdio.h>

#define  PI  3.141592654

double  area (double  r)
{
    return PI * r * r;
}

double  circumference (double  r)
{
    return 2.0 * PI * r;
}

double  volume (double r)
{
    return 4.0 / 3.0  *  PI * r * r * r;
}

int main (void)
{
    double  area (double  r), circumference (double  r),
            volume (double r);

    printf ("radius = 1: %.4f   %.4f   %.4f\n",
          area(1.0), circumference(1.0), volume(1.0));

    printf ("radius = 4.98: %.4f   %.4f   %.4f\n",
          area(4.98), circumference(4.98), volume(4.98));

    return 0;
}
```

Программа выдаст:

```
radius = 1: 3.1416   6.2832   4.1888
radius = 4.98: 77.9128   31.2903   517.3403
```

Директиву `#define` обычно пишут в начале программы, но это делается исключительно для упрощения чтения программы; его можно писать где угодно, лишь бы он стоял перед первым обращением к нему. Также константы `#define` не могут быть локальными, в отличие от обычных констант и переменных. Их всегда можно использовать **где угодно**.

Для обозначения нулевого указателя часто используется константа `NULL`:

```c
#define  NULL  0
```

так можно получить более читабельные выражения с указателями, вроде

```c
while ( listPtr != NULL )
        ...
```

Такой цикл будет выполняться, пока указатель `listPtr` не станет нулевым.

Ничто также не мешает определять выражения:

```c
#define  TWO_PI    2.0f * 3.14159f
```

Фактически, можно определять **что угодно**, компилятор подставит это вместо названия дефайна:

```c
#define  LEFT_SHIFT_8    << 8
```

```
x = y  LEFT_SHIFT_8;
```

Такой код вполне рабочий.

Можно еще так:

```
1    #define  AND   &&
2    #define  OR    ||
3
4    if ( x > 0  AND  x < 10 )
5        ...
```

Но хоть такие перестановки и могут кому-то упростить работу с кодом, изменение синтаксиса в языке считается плохой практикой, потому что другим людям может быть труднее работать с вашим кодом.

Константы также могут ссылаться друг на друга:

```
#define  PI        3.141592654
#define  TWO_PI    2.0 * PI
```

Кстати, наоборот тоже можно, это будет работать при условии, что константы объявлены до их использования в основном коде. Но так лучше не делать, потому что программу будет труднее читать:

```
#define  TWO_PI    2.0 * PI
#define  PI        3.141592654
```

Возможно, вы заметили, мы всегда писали названия констант **большими буквами**. Это просто общепринятый способ записи, чтобы сразу было понятно, что это константа из `#define`, т.е. ничто не мешает использовать строчные буквы при желании.

Расширяемость функционала программы

Использование `#define` упрощает модификацию функционала программы. Например, при объявлении массива нужно указывать количество элементов в нем и в большинстве случаев функции, работающие с этим массивом, должны учитывать количество элементов в нем. И если нужно изменить количество элементов массива, может понадобиться переписывать очень много других строк кода, где используется значение количества элементов. Иногда проще указать количество элементов через `#define`. Схематично программа с таким подходом будет выглядеть так:

```
1    #define  MAX_DATAVALUES   1000
2
3    float  dataValues[MAX_DATAVALUES];
4
5    for ( i = 0;  i < MAX_DATAVALUES;  ++i )
6        ...
7
8    if  ( index > MAX_DATAVALUES - 1 )
9        ...
```

Значение `MAX_DATAVALUES` можно поменять в любой момент и не нужно переписывать все строки, где оно используется.

Портируемость программ

Еще одно полезное применение `#define` – упрощение портирования кода с одной системы на другую. Это могут быть специфичные для разных систем адреса в памяти (привет, микроконтроллеры), разная длина машинного слова или разное количество памяти, выделяемое под переменные. Например, функция `rotate()` из урока 1.10 (*побитовые операции*) рассчитана только на системы,

где под тип `int` выделяется 32 бита; на системах, которые выделяют 64 бита под `int`, функция работать не будет. Проблему можно решить через `#define` и изменять константу в зависимости от системы, под которую будет компилироваться программа. Перепишем функцию `rotate()` с использованием директивы `#define`.

```
#include <stdio.h>

#define  kIntSize  32   // *** machine dependent !!! ***

// Function to rotate an unsigned int left or right

unsigned int  rotate (unsigned int  value, int  n)
{
    unsigned int  result, bits;

    /* scale down the shift count to a defined range */

    if  ( n > 0 )
        n = n % kIntSize;
    else
        n = -(-n % kIntSize);

    if  ( n == 0 )
        result = value;
    else if ( n > 0 )     /* left rotate */
    {
        bits = value >> (kIntSize - n);
        result = value << n  |  bits;
    }
    else                  /* right rotate */
    {
        n = -n;
        bits = value << (kIntSize - n) ;
        result = value >> n  |  bits;
    }

    return result;
}
```

Также можно написать функцию для определения выделяемого количества бит и код вообще не будет зависеть от этого, но иногда это непрактично и удобнее использовать именно `#define`.

Дополнительные способы применения констант

Константы можно использовать вместо коротких функций для лучшей компактности кода. Допустим, у нас есть выражение:

```
if ( year % 4 == 0  &&  year % 100 != 0  ||  year % 400 == 0 )
    ...
```

Оно определяет, високосный год или нет. Определим его через #define:

```
#define  IS_LEAP_YEAR    year % 4 == 0  &&  year % 100 != 0   \
                      || year % 400 == 0
    ...
if  ( IS_LEAP_YEAR )
    ...
```

Символ обратного слэша (\ ) используется для указания на перенос строки, потому что препроцессор ожидает, что выражение будет написано в одну строку и если есть перенос, нужно явно на это указывать.

Аргументы и макросы

Константу `IS_LEAP_YEAR` можно определить таким образом, чтобы она принимала аргумент `year`:

```
#define  IS_LEAP_YEAR(year)    y % 4 == 0  &&  y % 100 != 0  \
                            ||  y % 400 == 0
```

Между названием константы и скобками `(year)` **пробелы недопустимы**.

В отличии от объявления функции, здесь не нужно указывать возвращаемый тип данных, потому что здесь происходит просто подстановка текста туда, где будет вызвана константа. Вызов константы будет выглядеть так:

```
if ( IS_LEAP_YEAR (next_year) )
    ...
```

Подобные конструкции в C называются **макросами**. Основное их преимущество перед функциями – отсутствие необходимости определять тип входных и выходных данных. Например, у нас есть макрос

```
#define  SQUARE(x)   x * x

y = SQUARE (v); // usage example
```

который возводит свой аргумент в квадрат. Его можно использовать где угодно и вне зависимости от типа аргументов, будь это `int`, `long` или `float`. Функции же могут работать только с четко определенными типами данных. Недостаток макросов – итоговая программа будет занимать больше места, потому что макросы будут продублированы в ПЗУ столько раз, сколько они были использованы в программе, когда функция определяется в памяти только один раз. С другой стороны, процессору не нужно будет тратить время на переход в другой участок памяти, чтобы обратиться к функции.

Но есть один нюанс. Допустим, мы напишем

```
y = SQUARE (v + 1);
```

Это выражение не вычислит $(v + 1)^2$. Оно вычислит `v + 1 * v + 1`, потому что в макросах аргумент подставляется напрямую. Чтобы макрос корректно обработал подобный аргумент, в него нужно добавить скобки:

```
#define  SQUARE(x)  ( (x) * (x) )
```

и теперь выражение

```
y = SQUARE (v + 1);
```

будет корректно обработано как

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

В макросах еще иногда удобно применять тернарный оператор (*conditional expression operator*):

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

Этот макрос вернет больший из переданных аргументов. Скобки вокруг всего выражения нужны, чтобы правильно работали выражения вроде

```
MAX (x, y) * 100
```

А скобки вокруг каждого аргумента, чтобы были правильно обработаны выражения вроде

```
MAX (x & y, z)
```

Оператор побитового `AND` имеет более низкий приоритет, чем оператор `>` в макросе. Без скобок оператор `>` был бы выполнен до `AND`, что дало бы неправильный результат.

Следующий макрос проверяет, является ли символ строчной буквой:

```
#define  IS_LOWER_CASE(x)  ( ((x) >= 'a') && ((x) <= 'z') )
```

Этот же макрос можно применить в другом макросе, который конвертирует все строчные буквы в прописные:

```
#define  TO_UPPER(x) ( IS_LOWER_CASE (x) ? (x) - 'a' + 'A' : (x) )
```

Его можно использовать в цикле:

```
1    while ( *string  != '\0' )
2    {
3        *string = TO_UPPER (*string);
4        ++string;
5    }
```

который перебирает все символы строки и делает все строчные буквы в ней прописными.

прим.: в стандартных библиотеках C уже есть функции для проверки символов. Функции `islower` и `toupper` выполняют те же функции, что и описанные здесь макросы `IS_LOWER_CASE` и `TO_UPPER`. Обычно лучше пользоваться именно библиотечными функциями, а не изобретать велосипед и писать свои.

Передача переменного количества аргументов в макрос

Можно задать такой макрос, который будет принимать неопределенное количество аргументов. Это делается написанием многоточия в конец списка аргументов. Обращение к этим аргументам происходит через идентификатор `__VA_ARGS__`. Приведем в пример макрос `debugPrintf`:

```
#define debugPrintf(...)   printf ("DEBUG: "__VA_ARGS__);
```

Такой макрос можно использовать как обычный `printf()` (вспомните, что сама функция `printf()` точно так же принимает не один аргумент, а неопределенное количество аргументов):

```
1 debugPrintf ("Hello world!\n");
2 debugPrintf ("i = %i, j = %i\n", i, j);
```

Эти выражения выведут:

```
DEBUG: Hello world!
DEBUG: i = 100, j = 200
```

Оператор преобразования в строку # (*stringizing operator*)

Если перед параметром макроса поставить оператор #, то препроцессор превратит переданный параметр в строку-константу. Например, макрос

```
#define  str(x)   # x
```

при его вызове

```
str ("hello")
```

будет обработан препроцессором как

```
"\"hello\""
```

Да, вместе с кавычками, – не забываем про то, что препроцессор выполняет слепую подстановку.

Приведем более практичное применение оператора # – вывод на экран значения типа `int`:

```
#define  printint(var)   printf (# var " = %i\n", var)
```

Если count – переменная типа int, то выражение

```
printint (count);
```

превращается в

```
    printf ("count" " = %i\n", count);
```

А так как `printf()` последовательно выводит переданные в него строки, итоговым выражение будет

```
    printf ("count = %i\n", count);
```

Кстати, пробел между # и аргуметом при объявлении макроса не обязателен.

Оператор вставки токенов ## (*token-pasting operator*)

Этот оператор используется в макросах, чтобы соединить между собой два **токена**. Допустим, у нас есть список переменных от `x1` до `x100`. Можно сделать макрос `printx`, который будет получать на вход номер переменной `x` и печатать ее значение:

```
    #define  printx(n)   printf ("%i\n", x ## n)
```

Здесь выражение

```
    x ## n
```

фактически "склеивает" токены `x` и `n` в один токен (пробелы так же необязательны). То есть инструкция

```
    printx (20);
```

превратится в

```
    printf ("%i\n", x20);
```

Макрос `printx` можно также использовать в связке с макросом `printint` из предыдущего параграфа:

```
    #define  printx(n)   printint(x ## n)
```

Здесь инструкция

```
    printx (10);
```

сначала превратится в

```
    printint (x10);
```

затем в

```
    printf ("x10" " = %i\n", x10);
```

и наконец в

```
    printf ("x10 = %i\n", x10);
```

Непрактичный и сбивающий с толку пример. Придумать другой.

Директива #include

При создании программ на C вам скорее всего придется придется создавать много собственных макросов и функций, которые будут нужны в разных программах. Чтобы вам не приходилось каждый раз вписывать эти макросы и функции в свои программы, в C есть возможность добавить внешние файлы с макросами и функциями в программу. Этим и занимается директива `#include`. Обычно такие файлы имеют расширение `.h` и называются **файлами заголовка** (*header or include files*).

Допустим мы занимаемся метрологией и нам в разных программах нужно конвертировать значения. Список определений будет выглядеть примерно так:

```
    #define  INCHES_PER_CENTIMETER   0.394
    #define  CENTIMETERS_PER_INCH    1 / INCHES_PER_CENTIMETER

    #define  QUARTS_PER_LITER        1.057
    #define  LITERS_PER_QUART        1 / QUARTS_PER_LITER
```

```
#define  OUNCES_PER_GRAM         0.035
#define  GRAMS_PER_OUNCE         1 / OUNCES_PER_GRAM
   ...
```

Допустим, эти определения хранятся в файле `metric.h`. Если программе нужно использовать какое-либо определение из этого файла, этот файл нужно добавить в программу через директиву `#include`:

```
#include "metric.h"
```

Эта инструкция должна быть написана до того, как будут произведены какие-либо обращения к определениям из файла и потому ее обычно пишут в самом начале программы. Препроцессор ищет указанный файл в системе и фактически копирует его содержимое в то место программы, где была написана директива.

Обратите внимание, что кавычки здесь другие, не такие как были раньше. Если мы используем двойные кавычки (`""`), то препроцессор будет искать файл в локальной директории, – там, где находится исходный файл. Если же вместо кавычек используются `<>`, как мы делали раньше:

```
#include <stdio.h>
```

то препроцессор будет искать файл в системных директориях. В Windows эта директория зависит от используемой среды программирования. В Unix (включая Mac OS) это обычно `/usr/include`.

Напишем программу, которая будет использовать внешний файл с определениями `metric.h`.

```
1   /* Program to illustrate the use of the #include statement
2      Note: This program assumes that definitions are
3      set up in a file called metric.h             */
4
5   #include <stdio.h>
6   #include "metric.h"
7
8   int main (void)
9   {
10      float  liters, gallons;
11
12      printf ("*** Liters to Gallons ***\n\n");
13      printf ("Enter the number of liters: ");
14      scanf ("%f", &liters);
15
16      gallons = liters * QUARTS_PER_LITER / 4.0;
17      printf ("%g liters = %g gallons\n", liters, gallons);
18
19      return 0;
20  }
```

Для иллюстрации мы использовали только одно определение из файла, но ничто не мешает использовать все остальные.

Директива `#include` очень хороша тем, что позволяет централизовать все определения, – все программы будут использовать одни и те же значения. И если в одном из определений вдруг окажется ошибка, ее не нужно будет исправлять в каждой программе индивидуально.

Вообще в заголовочный файл можно добавлять фактически что угодно, не только определения. Там могут быть другие директивы препроцессора, структуры, глобальные переменные, прототипы функций, сами функции и т.д. Использование заголовочных файлов считается хорошим тоном в программировании. Также заголовочные файлы могут быть **вложенными**. То есть, один заголовочный файл может через `#include` ссылаться на другой файл, другой на третий и т.д.

Системные библиотеки (*System Include Files*)

Мы уже упоминали, что в файле `<stddef.h>` есть определение `NULL`, которое часто используется для проверки, является ли указатель нулевым. В `<math.h>` есть различные математические

константы вроде `M_PI`. В `<limits.h>` указаны максимальные диапазоны значений для различных типов переменных. Например, максимальное значение типа `int` указано в определении `INT_MAX`. В `<float.h>` также есть различные параметры для типов `float`, вроде максимальных значений (`FLT_MAX`) или количества цифр после запятой (`FLT_DIG`). Больше про системные библиотеки есть в приложении к курсу.

Условная компиляция (Conditional Compilation)

Условная компиляция используется, когда нужно написать программу, которая будет компилироваться на разных системах. Также ее можно использовать для включения/выключения различных элементов отладки, вроде вывода отладочных сообщений или отладочных переменных.

Директивы #ifdef, #endif, #else и #ifndef

В этом уроке мы демонстрировали способ улучшить портируемость функции `rotate()` через `#define`. Инструкция

```
#define  kIntSize  32
```

использовалась для устранения зависимости от количества бит, выделяемого под `unsigned int`. При этом упоминалось, что можно попросто написать функцию, которая автоматически будет определять выделяемое количество бит. Но такой способ не всегда прокатывает. Например, иногда работа программы зависит от структуры файловой системы, которая в разных ОС будет разная. Если это большая кроссплатформенная программа, скорее всего в ней будет очень много подобных зависимостей, специфичных для конкретной системы, которые может понадобиться менять для работы с каждой отдельной системой. Можно уменьшить себе геморрой, использовав директивы условной компиляции. Например, инструкции

```
1    #ifdef  UNIX
2    #   define  DATADIR    "/uxn1/data"
3    #else
4    #   define  DATADIR    "\usr\data"
5    #endif
```

присвоят `DATADIR` значение `"/uxn1/data"`, если символ `UNIX` был определен ранее, в ином случае будет присвоено `"\usr\data"`. Также обратите внимание на допустимость пробелов после #. Директивы `#ifdef`, `#else` и `#indef` говорят сами за себя. `#endif` заканчивает блок, начатый `#ifdef`, `#else` аналогичен обычному `else`.

Чтобы определить символ, достаточно инструкции

```
#define  UNIX    1
```

или просто

```
#define  UNIX
```

Многие компиляторы также разрешают задавать символы прямо в параметрах компиляции:

```
gcc -D UNIX program.c
```

После добавления символа любым из этих способов, все выражения `#ifdef UNIX` вернут `TRUE`.

Как избежать многократного добавления заголовочных файлов

Директива `#ifndef` аналогична `#ifdef`, но работает наоборот: возвращает `TRUE`, если указанный символ **не был** определен. Часто таким способом избегают многократного добавления заголовочных файлов. Это нужно для ускорения компиляции и избежания ошибок компиляции, связанных с многократным повторением одного и того же кода.

Допустим, у нас есть файл mystdio.h и для проверки, был ли файл добавлен или нет, мы прямо в этот

файл напишем строки

```
#ifndef _MYSTDIO_H
#define _MYSTDIO_H
  ...
#endif /* _MYSTDIO_H */
```

И теперь, после добавления файла в программу:

```
#include "mystdio.h"
```

директива `#ifndef` проверит, был ли уже определен `_MYSTDIO_H` или нет. Если нет, то будут выполнены строки между `#ifndef` и `#endif`.

Директивы #if и #elif

Эти директивы работают как обычные `if` и `else if` соответственно и могут применяться для проверки логических выражений. Например, мы пишем программу, которая будет работать под Windows, Linux и Mac OS. Создадим блок, который будет выполнять разные действия в зависимости от системы, для которой будет компилироваться программа:

```
#if    OS == 1  /* Mac OS */
  ...
#elif  OS == 2  /* Windows */
    ...
#elif  OS == 3  /* Linux  */
    ...
#else
    ...
#endif
```

В большинстве компиляторов значение `OS` можно задать через параметр `-D`, как мы уже делали:

```
gcc -D OS=2 program.c
```

Такая строка указывает, что компилируемая программа будет работать на Windows.

Также существует специальный оператор `defined()`:

```
defined (name)
```

который также может использоваться с директивой `#if`. Блоки

```
#if  defined (DEBUG)
    ...
#endif
```

и

```
#ifdef DEBUG
    ...
#endif
```

будут выполнять одно и то же действие.

Блок

```
1 #if defined (WINDOWS) || defined (WINDOWSNT)
2 #  define BOOT_DRIVE "C:/"
3 #else
4 #  define BOOT_DRIVE "D:/"
5 #endif
```

задаст `BOOT_DRIVE` значение `"C:/"`, если был определен `WINDOWS` или `WINDOWSNT` и `"D:/"` во всех остальных случаях.

Директива #undef

В некоторых случаях нужно "разопределить" определенный ранее символ. Этим и занимается директива `#undef`. Например, инструкция

```
#undef  LINUX
```

удалит определение `LINUX`. Все последующие команды `#ifdef LINUX` и `#if defined (LINUX)` будут возвращать `FALSE`. Если определения `LINUX` и так не существовало, директива просто не внесет никаких изменений.

Неясно практическое применение директивы.

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 12
2. Guarding against multiple inclusion of header files – http://www.keil.com/support/man/docs/armcc/armcc_chr1359124224501.htm
3. Preprocessor directives – https://docs.microsoft.com/en-us/cpp/preprocessor/preprocessor-directives
4. Token-pasting operator (##) – https://docs.microsoft.com/en-us/cpp/preprocessor/token-pasting-operator-hash-hash
5. #include directive (C/C++) – https://docs.microsoft.com/en-us/cpp/preprocessor/hash-include-directive-c-cpp
6. Nested Ternary Operator – https://www.geeksforgeeks.org/c-nested-ternary-operator/
7. Token pasting in C using a variable that increments – https://stackoverflow.com/questions/16216585/

Упражнения

2. *Locate the system header files <`stdio.h`>, <`limits.h`>, and <`float.h`> on your system. Examine the files to see what's in them.*

Файлы находятся в директории C:\Program Files (x86)\CodeBlocks\MinGW\include\. Как и предполагалось, т.к. пользуюсь средой Code::Blocks. С файлом `limits.h` все понятно, в остальных только частично разобрался, пока это выше моей компетенции.

3. *Define a macro `MIN` that gives the minimum of two values.*

```
1
2     #include <stdio.h>
3
4     #define MIN(a,b) ( ((a) < (b)) ? (a) : (b) )
5
6     int main (void)
7     {
8          int a = 10;
9          int b = 20;
10
11         printf ("%i\n", MIN(a, b));
12         return 0;
13    }
```

4. *Define a macro MAX3 that gives the maximum of three values.*

```
1
2 #include <stdio.h>
3
4 #define MAX3(a,b,c) (  ((a)>(b)) ? ((a)>(c)) ? (a) : (c) : ((b)>(c)) ? (b) : (c) )
```

```c

int main (void)
{
     int a = 25;
     int b = 35;
     int c = 60;

     printf ("%i\n", MAX3(a, b, c));
     return 0;
}
```

Тернарный оператор в макросе работает так:

```c
    if (a > b)
         if (a > c)
              return a
         else
              return c
    else if (b > c)
         return b
    else
         return c
```

5. *Write a macro SHIFT to perform the identical purpose as the `shift( )` function of Program 11.3.*

```c
#include <stdio.h>

#define SHIFT(value,n) ( ((n) > 0) ? ((value) << (n)) : ((value) >> (-n)) )

int main (void)
{
    unsigned int  w1 = 0177777u, w2 = 0444u;

    printf ("%o\t%o\n", SHIFT (w1, 5), w1 << 5);
    printf ("%o\t%o\n", SHIFT (w1, -6), w1 >> 6);
    printf ("%o\t%o\n", SHIFT (w2, 0), w2 >> 0);
    printf ("%o\n", SHIFT (SHIFT (w1, -3), 3));

    return 0;
}
```

Забавно, насколько короче становится код. Но хуже читаемый для новичка, ага.

6. *Write a macro `IS_UPPER_CASE` that gives a nonzero value if a character is an uppercase letter.*
7. *Write a macro `IS_ALPHABETIC` that gives a nonzero value if a character is an alphabetic character. Have the macro use the `IS_LOWER_CASE` and the `IS_UPPER_CASE` macro.*
8. *Write a macro `IS_DIGIT` that gives a nonzero value if a character is a digit '0' through '9'. Use this macro in the definition of another macro `IS_SPECIAL`, which gives a nonzero result if a character is a special character; that is, not alphabetic and not a digit. Be certain to use the `IS_ALPHABETIC` macro.*
11. *Test the system library functions that are equivalent to the macros you developed in the preceding three exercises. The library functions are called `isupper`, `isalpha`, and `isdigit`.*

```c
#include <stdio.h>
#include <ctype.h>

#define  IS_UPPER_CASE(x)  ( (((x) >= 'A') && ((x) <= 'Z')) ? 1 : 0 )
#define  IS_LOWER_CASE(x)  ( (((x) >= 'a') && ((x) <= 'z')) ? 1 : 0 )
#define  IS_ALPHABETIC(x)  ( (IS_UPPER_CASE(x) || IS_LOWER_CASE(x)) ? 1 : 0 )
#define  IS_DIGIT(x)       ( (((x) >= '0') && ((x) <= '9')) ? 1 : 0 )
```

```c
#define  IS_SPECIAL(x)     ( (IS_ALPHABETIC(x) || IS_DIGIT(x)) ? 0 : 1 )

int main (void)
{
     char a;
     scanf("%c", &a);
     printf("Upper: %i\n", IS_UPPER_CASE(a));
     printf("Lower: %i\n", IS_LOWER_CASE(a));
     printf("Alpha: %i\n", IS_ALPHABETIC(a));
     printf("Digit: %i\n", IS_DIGIT(a));
     printf("Specl: %i\n", IS_SPECIAL(a));

     printf("\nTest Upper: %i\n", isupper(a));
     printf("Test Alpha: %i\n", isalpha(a));
     printf("Test Digit: %i\n", isdigit(a));

     return 0;
}
```

Интересное наблюдение, что `isalpha()` возвращает 2, если буква строчная и 1, если прописная.

9. *Write a macro ABSOLUTE_VALUE that computes the absolute value of its argument.*

```c
#include <stdio.h>

#define  ABSOLUTE_VALUE(x)  ( ((x) >= 0) ? x : -(x) )

int main (void)
{
     int a;
     scanf("%i", &a);
     printf("ABS: %i\n", ABSOLUTE_VALUE(a + 10));

     return 0;
}
```

10. *Could the following be used to display the values of the 100 variables x1–x100? Why or why not?*

```c
#define printx(n)  printf ("%i\n", x ## n)
for (i = 1; i < 100; ++i)
     printx (i);
```

У меня ничего не получилось. Скорее всего потому что оператор вставки токенов к букве `x` прилепляет саму букву `i`, а не значение, которому равна переменная `i`. И вообще, лучше использовать массивы, а не индексированные переменные.