ШАГ компьютерная академия

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С

Урок №18

Программирование на языке C

Содержание

1. Препроцессор

Препроцессор — это программа, которая производит некоторые (иногда весьма значительные) манипуляции с первоначальным текстом программы перед тем, как он подвергается компиляции. Будучи дословно переведенным, с английского, слово **препроцессор** означает **предварительный обрабатыватель**.

Препроцессоры создают входной текст для компиляторов и могут выполнять следующие функции:

- обработку макроопределений;
- включение файлов;
- «рациональную» предобработку;
- расширение языка.

Например, весьма часто в программах приходится использовать «ничего не говорящие» числа. Это могут быть какие-то математические константы или размеры используемых в программе массивов и т.д. Общепризнано, что обилие таких констант затрудняет понимание программ и считается признаком плохого стиля программирования. В среде программистов такие константы получили язвительное название **магических чисел**. Чтобы программа не изобиловала ими, языки программирования позволяют дать константе **имя** и далее использовать его везде вместо самой константы.

В языке С такую возможность обеспечивает препроцессор. Например, с помощью определений

```
#define P1 3.14159
#define E  2.71284
```

препроцессор заменит в программе все имена P1 и E на соответствующие числовые константы. Теперь, когда вы обнаружите, что неправильно написали приближенное значение основания натуральных логарифмов, вам достаточно исправить единственную строку с определением константы, а не просматривать всю программу:

```
#define E 2.71828
```

Препроцессор языка С позволяет переопределять не только константы, но и целиком программные конструкции. Например, можно написать определение:

```
#define forever for(;;)
```

и затем всюду писать бесконечные циклы в виде:

```
forever { <тело цикла> }
```

А если вам не нравятся фигурные скобки, то определите

```
#define begin {
#define end }
```

и далее используйте в качестве операторных скобок **begin** и **end**, как это делается, например, в языке **Pascal**. Подобные определения, называемые **макроопределениями** (макросами), могут иметь параметры (и вследствие этого быть еще более мощными), однако об этом чуть позже.

Еще одна важная «услуга» препроцессора — включение в исходный текст содержимого других файлов. Эта возможность в основном используется для того, чтобы снабжать программы какими-то общими для всех файлов определениями. Например, чрезвычайно часто в начале программы на языке С встречается препроцессорная конструкция:

```
#include <iostream>
```

Когда исходный текст программы обрабатывается препроцессором, на место этой инструкции ставится содержимое файла **iostream**, содержащего макроопределения и объявления данных, необходимых для работы потоков ввода-вывода.

Оператор (директива) препроцессора — это одна строка исходного текста, начинающаяся с символа #, за которым следуют название оператора (**define, pragma, include, if**) и операнды. Операторы препроцессора могут появляться в любом месте программы, и их действие распространяется на весь исходный файл.

2. Определение констант с помощью #define

Оператор **#define** часто используют для определения **символических констант**. Он может появиться в любом месте исходного файла, а даваемое им определение имеет силу, начиная с места появления и до конца файла.

Примечание: В конце определения символической константы (в конце оператора #define) точка с запятой не ставится!

```
# define min 1
# define max 100
```

В тексте программы вместо констант 1 и 100 можно использовать соответственно **min** и **max**.

```
#include <iostream>
using namespace std;
#define NAME "Vasya Pupkin."
void main ()
{
    cout << " My name is " << NAME;
}
Результат работы:
My name is Vasya Pupkin.
```

Примечание: Текст внутри строк, символьные константы и комментарии не подлежат замене, т.к. строки и символьные константы являются неделимыми лексемами языка С. Так что, после макроопределения

```
#define YES 1
```

в операторе

```
cout << "YES";
```

не будет сделано никакой макроподстановки.
Замены в тексте можно отменять с помощью команды:

```
#undef <имя>
```

После выполнения такой директивы имя для препроцессора становится неопределенным и его можно определять повторно. Например, не вызовут предупреждающих сообщений директивы:

```
#define M 16
#undef M
#define M 'C'
#undef M
#define M "C"
```

Директиву **#undef** удобно использовать при разработке больших программ, когда они собираются из отдельных «кусков текста», написанных в разное время или разными программистами. В этом случае могут встретиться одинаковые обозначения разных объектов. Чтобы не изменять исходных файлов, включаемый текст можно «обрамлять» подходящими директивами **#define — #undef** и тем самым устранять возможные ошибки. Например:

```
.    .    .    .    .
     A = 10; //Основной текст.
   .    .    .    .    .
   #define A X
   .    .    .    .    .
     A = 5; //Включенный текст.
   .    .    .    .    .
   #undef A
   .    .    .    .    .
     B = A; //Основной текст.
   .    .    .    .    .
```

При выполнении программы **B** примет значение 10, несмотря на наличие оператора присваивания **A=5**; во включенном тексте.

Если **строка_лексем** оказывается слишком длинной, то ее можно продолжить в следующей строке текста программы. Для этого в конце продолжаемой строки помещается символ «\». В ходе одной из стадий препроцессорной обработки этот символ вместе с последующим символом конца строки будет удален из программы. Например:

```
#define STROKA "\n Multum, non multa - \
    mnogoe, no nemnogo!"
   .    .    .    .    .
   cout << STROKA;
   .    .    .    .    .

На экран будет выведено:

   Multum, non multa - mnogoe, no nemnogo!
```

Напоминаем вам, что с помощью директивы #define мы с вами также создавали макросы, когда изучали встраивание в уроке номер девять. Рекомендуем Вам вернуться к этому уроку и повторить пройденный материал.

3. Условная компиляция

Директивы условной компиляции, позволяют генерировать программный код в зависимости от выполнимости определенных условий. Условная компиляция обеспечивается в языке С набором команд, которые, по существу, управляют не компиляцией, а препроцессорной обработкой:

```
#if <константное_выражение>
#ifdef <идентификатор>
#ifndef <идентификатор>
#else#endif#elif
```

Первые три команды выполняют проверку условий, две следующие — позволяют определить диапазон действия проверяемого условия. Последняя команда используется для организации проверки серии условий. Общая структура применения директив условной компиляции такова:

```
#if/#ifdef/#ifndef <константное_выражение или идентификатор>
          <текст_1>
#else //необязательная директива
          <текст_2>
#endif
```

- Конструкция **#else <текст_2>** не обязательна.
- **Текст_1** включается в компилируемый текст только при истинности проверяемого условия.
- Если условие ложно, то при наличии директивы **#else** на компиляцию передается **текст_2**.
- Если директива **#else** отсутствует, то весь текст от **#if** до **#endif** при ложном условии опускается.

Различие между формами команд **#if** состоит в следующем.

1. В первой из перечисленных директив **#if** проверяется значение константного целочисленного выражения. Если оно отлично от нуля, то считается, что проверяемое условие истинно. Например, в результате выполнения директив:

```
#if 5+12
       <текст_1>
#endif
```

текст_1 всегда будет включен в компилируемую программу.

2. В директиве **#ifdef** проверяется, определен ли с помощью команды **#define** к текущему моменту идентификатор, помещенный после **#ifdef.** Если идентификатор определен, то текст_1 используется компилятором.

3. В директиве **#ifndef** проверяется обратное условие - истинным считается неопределенность идентификатора, т.е. тот случай, когда идентификатор не был использован в команде **#define** или его определение было отменено командой **#undef.**

Для организации мульти ветвлений во время обработки препроцессором исходного текста программы введена директива

```
#elif <константное_выражение>
```

является сокращением конструкции #else#if.

Структура исходного текста с применением этой директивы такова:

```
#if <константное_выражение_1>
          <текст_1>
     #elif <константное_выражение_2>
          <текст_2>
     #elif <константное_выражение_3>
          <текст_3>
     .    .    .    .
     #else
          <текст_N>
     #endif
```

- Препроцессор проверяет вначале условие в директиве **#if**, если оно ложно (равно 0) — вычисляет **константное_выражение_2**, если оно равно О — вычисляется **константное_выражение_3** и т.д.
- Если все выражения ложны, то в компилируемый текст включается текст для случая **#else**.
- В противном случае, т.е. при появлении хотя бы одного истинного выражения (в **#if** или в **#elif**), начинает обрабатываться текст, расположенный непосредственно за этой директивой, а все остальные директивы не рассматриваются.
- Таким образом, препроцессор обрабатывает всегда только один из участков текста, выделенных командами условной компиляции.

А, теперь, рассмотрим несколько примеров.

Пример 1. Простая директива условного включения.

```
#ifdef ArrFlg
      int Arr[30];
   #endif
```

Если во время интерпретации директивы определено макроопределение ArrFlg, то приведенная запись дает генерацию выражения

```
   int Arr[30];
```

В противном случае не будет генерировано ни одно выражение.

Пример 2.

```
#include <iostream>
using namespace std;
#define ArrFlg 1
void main ()
{
   #ifdef ArrFlg
       int Arr[30];
   #else
       cout << "Array is not defined!";
   #endif
}
```

Пример 3. Директива условного включения с альтернативой

```
#if a+b==5
      cout << 5;
   #else
      cout << 13;
   #endif
```

Если выражение a+b==5 представляет величину, отличную от 0, то будет сгенерирована команда cout << 5;, в противном случае будет сгенерирована команда cout << 13;.

Пример 4. Составная директива условного включения

```
#include <iostream>
using namespace std;
//++++++++++++++++++++
#define Alfa 5
//++++++++++++++++++++
#if Alfa*5>20
      void main ()
        //++++++++++++++++++++
        #if Alfa==4
                int Arr[2];
        #elif Alfa==3
                char Arr[2];
        #else
                {
        #endif
        //++++++++++++++++++++
        #if 0
                cout<<"One";
        #else
                cout<<"Two";
        #endif
//++++++++++++++++++++
#else
   cout<<"Test";
#endif
//++++++++++++++++++++
}
```

Интерпретация приведенной записи приведет к генерации

```
void main ()
   {
      cout<<"Kaja";
   }
```

4. Другие директивы препроцессора

Кроме уже известных нам, существует несколько дополнительных директив, вот некоторые из них:

1. Для нумерации строк можно использовать директиву:

```
#line <константа>
```

которая указывает компилятору, что следующая ниже строка текста имеет номер, определяемый целой десятичной константой. Команда может определять не только номер строки, но и имя файла:

```
#line <константа> "<имя_файла>"
```

2. Директива

```
#error <последовательность_лексем>
```

приводит к выдаче диагностического сообщения в виде последовательности лексем. Естественно применение директивы **#error** совместно с условными препроцессорными командами. Например, определив некоторую препроцессорную переменную **NAME**

```
#define NAME 5
```

в дальнейшем можно проверить ее значение и выдать сообщение, если у NAME другое значение:

```
#if (NAME != 5)
#error NAME должно быть равно 5!
```

Сообщение будет выглядеть так:

```
fatal: <имя_файла> <номер_строки>
#error directive: NAME должно быть равно 5!
```

3. Команда

```
#pragma <последовательность_лексем>
```

определяет действия, зависящие от конкретной реализации компилятора, и позволяет выдавать компилятору различные инструкции.

4. В языке C существует возможность работы с операторами # и ##. Данные операторы используются в альянсе с директивой #define.

- Оператор # превращает аргумент, которому он предшествует, в строку, заключенную в кавычки.

```
#include <iostream>
using namespace std;
# define mkstr(s) #s
void main()
{
        cout<<mkstr(I love C);
        // Для компилятора cout<<"I love C";
}
```

- Оператор ## используется для конкатенации (объединения) двух лексем

```
#include <iostream>
using namespace std;
# define concat(a,b) a##b
void main()
{
        int xy=10;
        cout<<concat(x,y);
        // Для компилятора cout<<xy;
}
```

5. Разнесение проекта по несколько файлов

Как Вы уже давно знаете, для включения текста из файла используется команда #include. Пора познакомиться с ней поближе. Эта команда является директивой препроцессора и имеет две формы записи:

```
#include <имя_файла>// Имя в угловых скобках.
#include "имя_файла"// Имя в кавычках.
```

Если имя_файла — в угловых скобках, то препроцессор разыскивает файл в стандартных системных каталогах. Если имя_файла заключено в кавычки, то вначале препроцессор просматривает текущий каталог пользователя и только затем обращается к просмотру стандартных системных каталогов.

Начиная работать с языком С, мы сразу же столкнулись с необходимостью использования в программах средств ввода-вывода. Для этого в начале текста программы мы размещали директиву:

```
#include <iostream>
```

Выполняя эту директиву, препроцессор включает в программу средства связи с библиотекой ввода-вывода. Поиск файла **iostream** ведется в стандартных системных каталогах.

Заголовочные файлы оказываются весьма эффективным средством при модульной разработке крупных программ. Также, в практике программирования на С обычна ситуация, при которой, если в программе используется несколько функций, то удобно тексты этих функций хранить в отдельном файле. При подготовке программы пользователь включает в нее тексты используемых функций с помощью команд **#include**.

В качестве примера рассмотрим задачу обработки строк, в которой используем функции обработки строк, тексты которых находятся в отдельном файле.

Пример программы.

Ввести с клавиатуры заканчивающееся точкой предложение, слова в котором отделены друг от друга пробелами. Записать каждое слово предложения в обратном порядке (инвертировать слово) и напечатать полученное предложение. Для простоты реализации ограничим длину вводимого предложения 80 символами. Тогда программа решения сформулированной задачи может быть такой:

Основной файл:

```
#include <iostream>
using namespace std;
// Файлйл написанный самостоятельно, содержащий
// функцию соединения строк и
// функцию инвертирования строк.ы
#include "mystring.h"

void main()
{
   char slovo[81], sp[81], c = ' ', *ptr = slovo;
```

```
   sp[0] = '\0'; // Очистка массива для нового предложения.

   cout << "Enter string with point of the end:\n";
   do
   {
      cin >> slovo ; // Читается слово из входного потока.
       invert(slovo); // Инвертировать слово.
       c = slovo[0];

       // Убрать точку в начале последнего слова.
       if (c == '.')
                   ptr = &slovo[1];

       if (sp[0] != '\0')
            conc(sp," \0"); // Пробел перед словом.
      conc(sp,ptr);        // Добавить слово в предложение.

    }while (c != '.');     // Конец цикла чтения.

   conc(sp,".\0");          // Точка о конце предложения.
   cout << "\n" << sp;      // Вывод результата.
}
```

Заголовочный файл mystring.h:

```
void invert (char *e)
{
   char s;
   for (int m=0;e[m]!='\0';m++);
   for (int i=0,j=m-1;i < j;i++,j--)
   {
           s = e[i];
           e[i] = e[j];
           e[j] = s;
   }
}

void conc (char *c1, char *c2)
{
   for (int m=0;c1[m]!='\0';m++);
   // m - длина первой строки.
   for (int i=0; c2[i]!='\0';i++)
      c1[m+i]=c2[i];
   c1[m+i] = '\0';
}
```

Комментарий к коду.

- В программе в символьный массив slovo считывается из входного потока (с клавиатуры) очередное слово.
- sp — формируемое предложение, в конец которого всегда добавляется точка.
- Переменная char c — первый символ каждого инвертированного слова.
- Для последнего слова предложения этот символ равен точке.
- При добавлении этого слова к новому предложению точка отбрасывается с помощью изменения значения указателя ptr.
- Использованы директивы #include, включающие в программу средства ввода/вывода и тексты функций инвертирования строки invert() и конкатенации строк conc().
- Обратите внимание, что длина массива — первого из параметров функции conc() должна быть достаточно велика, чтобы разместить результирующую строку.
- Препроцессор добавляет тексты всех функций в программу из файла mystring.h и как единое целое передает на компиляцию.

Примечание: Кстати, для того, что бы добавить к проекту заголовочный файл, необходимо произвести все те же действия, что и при добавлении основного файла, но при выборе типа файла следует остановиться на шаблоне Header File (.h)

6. Экзаменационные задания

1. Создать программу, фильтрующую текст, введенный с клавиатуры. Задача программы заключается в считывании текста и отображении его на экране , используя замену заданного набора символов на пробелы. Программа должна предлагать следующие варианты наборов символов для фильтрации:

- Символы латинского алфавита
- Символы кириллицы
- Символы пунктуации
- Цифры

Фильтры могут накладываться последовательно. При повторной установке существующего фильтра данный фильтр должен сниматься.

2. Написать «Морской бой» для игры человека против компьютера. Предусмотреть за человека возможность автоматической (расстановку осуществляет кораблей компьютер случайным образом) и ручной расстановки своих кораблей. Стоимость задания существенно повышается, если компьютер при стрельбе будет обладать логикой (т. е. не производить выстрелы «рандомайзом»).

3. Создать приложение для вычисления значения арифметического выражения, которое может включать в себя действительные числа, а также круглые скобки и следующие операции: +, –, *, /, ^ (возведение в степень). Вычисления должны производиться с учетом скобок и приоритетов используемых операций. Предусмотреть корректную обработку возможных ошибок и информирование о них пользователя.