ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИЙ И ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

НУКУССКИЙ ФИЛИАЛ ТАШКЕНТСКОГО УНИВЕРСИТЕТА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Курсовая работа

По предмету Программирование на C++

На тему: ***Макросы***

Студента 2в курса направления «Телекоммуникационные технологии»

Длимбетов А.

Преподаватель: Ядгаров Ш.

Нукус 2015

План

Введение

Независимо от используемой операционной системы и программных приложений пользователь часто выполняет одни и те же последовательности команд для многих рутинных задач. Вместо повторения последовательности команд каждый раз, когда необходимо выполнить какую-либо задачу, можно создать макрос, который будет выполнять эту последовательность. Макросы позволяют вводить одиночную команду, выполняющую ту же задачу, для реализации которой было бы необходимо вводить несколько команд вручную.

Записанные макрорекордером последовательности команд первоначально назывались макрокомандами. Сейчас этот термин сократился до более простого слова - макрос. Применительно к информатике и программным приложениям под словом макрос всегда подразумевается макрокоманда.

Макросы, кроме удобства, имеют и другие преимущества. Поскольку компьютеры больше приспособлены для выполнения повторяющихся задач, чем люди, запись макрорекордером неоднократно выполняемых команд повышает точность и скорость работы. Другим преимуществом использования макросов является то, что при их выполнении обычно нет необходимости в присутствии человека-оператора. В случае, если макрос очень длинный или выполняет операции, требующие значительного времени, можно оставить работающий компьютер и делать что-нибудь другое, или переключиться на другое приложение.

Макрорекордер (или просто "рекордер") записывает все действия пользователя, включая ошибки и неправильные запуски. Когда программа воспроизводит макрос, она выполняет каждую записанную рекордером команду точно в такой последовательности, в которой она выполнялась во время записи. Первые макрорекордеры имели серьезный недостаток. Если во время записи длинной последовательности действий была допущена ошибка, то единственной возможностью удалить эту ошибку являлась повторная запись макроса. Кроме того, если надо было внести небольшое изменение в длинный макрос, то также приходилось перезаписывать весь макрос. Перезапись длинного макроса часто приводила к дополнительным ошибкам в новой записи. По этим причинам разработчики программного обеспечения добавили макрорекордерам возможность редактирования макросов, чтобы можно было легко исправлять небольшие ошибки или вносить другие изменения в макрос без его полной перезаписи.

Ключевое слово Sub. Этим обозначается начало макроса. Из-за ключевого слова Sub (от анг subroutine - подпрограмма) командные макросы также называются процедурами-подпрограммами.

Имя макроса. После ключевого слова Sub Excel добавляет имя макроса, за которым следует открывающаяся и закрывающаяся скобки.

Комментарии. Первые несколько строк кода начинаются с апострофа """, которые говорят редактору, что эти строки являются комментариями. Комментарии отображаются только в окне редактора, при выполнении макроса они не обрабатываются. В каждом записанном макросе в комментариях указывается имя макроса, а также описание, которые вы ввели в диалоговом окне "Запись макроса".

Макрооператоры. Основное тело макроса (другими словами, строки между ключевыми словами Sub и End Sub, исключая комментарии в начале макроса) состоит из последовательности операторов. Они являются интерпретацией действий, которые вы выполнили во время записи макроса.

Строки макроса в модуле C++ являются обычным текстом, который можно изменять также, как это делается в любом текстовом редакторе. Если макрос содержит операторы, которые необходимо удалить, можно просто удалить лишние строки из модуля.

Часто нужно добавить новые действия в записанный макрос. К сожалению, C++ не предоставляет никаких возможностей записи новых операторов в существующий макрос. Вместо этого, нужно сначала записать новый макрос, содержащий необходимые команды, и отобразить его код на экране. Затем можно использовать стандартные средства Windows копирования и вставки (можно просто перетащить текст из одного окна в другое), чтобы перенести необходимые операторы из нового макроса с исходный.

В языке ассемблера, а также в некоторых других языках программирования, макрос — символьное имя, заменяемое при обработке препроцессором на последовательность программных инструкций. Для каждого компилятора (ассемблера) существует специальный синтаксис объявления и вызова макросов. При этом внутри макроса могут быть условные операторы препроцессора, многие компиляторы поддерживают при вызове макросов передачу аргументов. В этом случае один и тот же макрос может «разворачиваться» в различные последовательности инструкций при каждом вызове — в зависимости от сработавших разветвлений внутри макроса и переданных ему аргументов.

Что такое макрос?

Приходилось ли вам при работе с программой Microsoft Office выполнять повторяющиеся задачи, которые можно было бы выполнять автоматически? Возможно, вам приходилось переформатировать многочисленные таблицы в документе Word или преобразовывать данные в электронных таблицах Excel. А может, было необходимо сделать однотипные изменения на нескольких похожих страницах Visio или слайдах PowerPoint? Если описанные ситуации кажутся вам знакомыми, значит, вам пора узнать больше о макросах.

Возможно, вы когда-нибудь работали с файлами, открытие которых сопровождалось предупреждением безопасности о том, что в файлах содержатся макросы, поэтому макросы чаще всего ассоциируются с пугающими словами «вирус» или «программирование». Запомните, что большинство макросов не только безвредны, но и могут значительно экономить ваше время. Кроме того, создать макрос намного проще, чем это может вам показаться.

Не имеет значения, в каком приложении пакета Office вы работаете - это может быть Word, Excel или PowerPoint - вероятно вы по нескольку по нескольку раз в день выполняете некоторые рутинные операции, которые вы либо уже десятки раз выполняли раньше, либо которые вам нужно повторить много раз. В частности, это форматирование отдельных фрагментов текста, последовательное выполнение нескольких команд меню или форматирование документа определенным образом. Я думаю, что вы мечтаете избавиться от этой рутины и сократить время, необходимое на выполнение работы.

Конечно, большинство приложений пакета Office содержит в меню Правка команду повторить, с помощью которой можно повторить последнее действие. Это очень удобно, но с помощью этой команды можно повторить только одно действие. Если же нужно повторить несколько действий, то данная команда не подойдет.

Что следует делать в таких случаях?

Можно автоматизировать практически любую рутинную работу и повторяющуюся работу. Можно выполнить эту автоматизированную работу МГНОВЕННО, просто выбрать одну команду, нажав клавишу либо щелкнув на панели инструментов.

Это звучит слишком хорошо, чтобы быть правдой, но, используя VBA, можно сделать нечто, называемое МАКРОСОМ. Он в действительности состоит из списка действий, которые надо запомнить для повторного выполнения. Таким образом, макрос не сильно отличается от кулинарного рецепта, состоящего из набора инструкций, в которых говориться, какие действия необходимо выполнить, чтобы что-нибудь приготовить.

Макрос - это набор инструкций, которые сообщают программе (такой как Word или Excel), какие действия следует выполнить, чтобы достичь определенной цели.

Различие, однако, состоит в том, что макрос объединяет все эти инструкции в одном сценарии, который затем можно вызвать с помощью команды меню, кнопки панели инструментов или комбинации клавиш. С этой точки зрения макрос отличается от, скажем, рецепта приготовления хлеба, но похож на автоматическую хлебопекарню, загрузив ингредиенты в которую, можно испечь хлеб одним нажатием кнопки.

Список инструкций, составляющих макрос, как правило, состоит из макрооператоров. Некоторые операторы выполняют особые действия, связанные с выполнением самого макроса, но большинство операторов соответствует командам меню и опциям диалоговых окон приложения, в котором выполняется макрос.

Основное назначение макросов — автоматизация работы пользователя. Кроме этого, созданный код макроса может служить основой для дальнейших разработок.

При записи макроса запоминаются все действия пользователя, будь то нажатие клавиши или выбор определенной команды меню, которые автоматически преобразуются в программный код на языке C++. Каждому макросу дается имя, а для быстрого запуска макроса можно создать или присвоить ему “горячую” клавишу (клавишу, по нажатию на которую будет производиться запуск макроса). После запуска макрос будет автоматически выполнен тем приложением, в котором он создан и запущен. При выполнении макроса компьютер воспроизведет все действия пользователя. Макрос — это именованная последовательность заданных пользователем команд и действий, хранящаяся в форме программы на языке C++.

Когда и зачем использовать макросы?

Макросы экономят время и расширяют возможности ежедневно используемых программ. Макросы можно использовать для автоматизации выполнения повторяющихся действий при редактировании документа, оптимизации выполнения трудоемких задач и для создания решений, например для автоматизации создания документов, которые вы и ваши коллеги постоянно используете. Те, кто хорошо знаком с языком VBA, могут использовать макрос для создания пользовательских надстроек, включающих шаблоны, диалоговые окна, и даже для хранения многократно использующихся сведений.

Макросы и безопасность

В то время как большинство макросов не только безвредны, но и полезны, макросы представляют собой важную проблему безопасности. Макрос, созданный с вредительскими целями, может содержать код, который повлечет повреждение или уничтожение документа и даже всей системы.

Чтобы защитить систему и файлы, не включайте макросы из неизвестных источников. Чтобы иметь возможность включать и отключать

макросы и при этом иметь доступ ко всем макросам, которые необходимо использовать, в приложениях семейства Office установите средний уровень безопасности. Тогда вы получите возможность включать или выключать макросы каждый раз при открытии файла, который содержит макрос, и при этом сможете запускать любой макрос по вашему выбору.

Макросы в C++

Простые макросы

"Простой макрос" это тип сокращения. Это идентификатор, который используется для представления фрагмента кода.

Перед использованием макроса его необходимо определить с помощью директивы '#define', за которой следует название макроса и фрагмент кода, который будет идентифицировать этот макрос. Например,

```
#define BUFFER_SIZE 1020
```

определяет макрос с именем 'BUFFER_SIZE', которому соответствует текст '1024'. Если где-либо после этой директивы встретится выражение в следующей форме:

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

то C препроцессор определит и заменит макрос 'BUFFER_SIZE' на его значение и в результате получится

```
foo = (char *) xmalloc (1020);
```

Использование прописных букв в названиях макросов является стандартным соглашением и повышает читабельность программ.

Обычно, макроопределением должна быть отдельная строка, как и при использовании всех директив препроцессора. (Длинное макроопределение можно разбить на несколько строк с применением последовательности backslash-newline.) Хотя существует одно исключение: символы перевода строки могут быть вкючены в макроопределение если

они находятся в строковой или символьной константе, потому как макроопределение не может содержать каких-либо специальных символов. Макроопределение автоматически дополняется соответствующим специальным символом, который завершает строчную или символьную константу. Комментарии в макроопределениях могут содержать символы перевода строки, так как это ни на что не влияет, потому как все комментарии полностью заменяются пробелами вне зависимости от того, что они содержат.

В отличие от выше сказанного, не существует никаких ограничений на значение макроса. Скобки не обязательно должны закрываться. Тело макроса не обязательно должно содержать правильный С код.

Препроцессор С++ обрабатывает программу последовательно, поэтому макроопределения вступают в силу только в местах, где они используются. Поэтому, после обработки следующих данных С препроцессором

```
foo = X;
#define X 4
bar = X;
```

получится такой результат

```
foo = X;
bar = 4;
```

После подстановки препроцессором имени макроса, тело макроопределения добавляется к началу оставшихся вводимых данных и происходит проверка на продолжение вызовов макросов. Поэтому тело макроса может содержать ссылки на другие макросы. Например, после выполнения

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

значением макроса 'TABLESIZE' станет в результате значение '1020'.

Это не является тем же, что и определение макроса 'TABLESIZE'

равным значению '1020'. Директива '#define' для макроса 'TABLESIZE' использует в точности те данные, которые были указаны в ее теле и заменяет макрос 'BUFSIZE' на его значение.

Макросы с аргументами

Значение простого макроса всегда одно и то же при каждом его использовании. Макросы могут быть более гибкими, если они принимают аргументы. Аргументами являются фрагменты кода, которые прилагаются при каждом использовании макроса. Эти фрагменты включаются в расширение макроса в соответствии с указаниями в макроопределении.

Для определения макроса, использующего аргументы, применяется директива '#define' со списком имен аргументов в скобках после имени макроса. Именами аргументов могут быть любые правильные C идентификаторы, разделенные запятыми и, возможно, пробелами. Открывающаяся скобка должна следовать сразу же после имени макроса без каких-либо пробелов.

Например, для вычисления минимального значения из двух заданных можно использовать следующий макрос:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

Для применения макроса с аргументами нужно указать имя макроса, за которым следует список аргументов, заключенных в скобки и разделенных запятыми. Количество принимаемых аргументов должно соответствовать количеству указываемых. Например, макрос 'min' можно использовать так: 'min (1, 2)' или 'min (x + 28, *p)'.

Значение макроса зависит от используемых аргументов. Каждое имя аргумента во всем макроопределении заменяется на значения соответствующих указанных аргументов. При использовании макроса 'min', рассмотренного ранее, следующим образом:

```
min (1, 2)
```

будет получен следующий результат:

```
((1) < (2) ? (1) : (2))
```

где '1' заменяет 'X', а '2' заменяет 'Y'.

При указании аргументов, скобки должны закрываться, а запятая не должна завершать аргумент. Однако, не существует каких либо ограничений на использование квадратных или угловых скобок. Например

```
macro (array[x = y, x + 1])
```

передает макросу 'macro' два аргумента: 'array[x = y' и 'x + 1]'.

После подстановки указанных аргументов в тело макроса, полученный в результате текст добавляется к началу оставшихся данных и производится проверка на наличие других вызовов макросов. Поэтому указываемые аргументы могут содержать ссылки к другим макросам как с аргументами, так и без, а также к тому же макросу. Тело макроса также может включать ссылки к другим макросам. Например, макрос 'min (min (a, b), c)' заменяется следующим текстом:

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(Срока разбита на три для ясности и в действительности она не разбивается.)

Если макрос 'foo' принимает один аргумент и нужно передать ему пустой аргумент, то в скобках следует указать по крайней мере один пробел: 'foo ( )'. Если пробел не указывать, а макрос 'foo' требует один аргумент, то произойдет ошибка. Для вызова макроса, не принимающего аргументы, можно использовать конструкцию 'foo0()' как рассмотрено ниже:

```
#define foo0() ...
```

Если используется имя макроса, за которым не следует открывающаяся скобка (после удаления всех следующих пробелов,

символов табуляции и комментариев), то это не является вызовом макроса и препроцессор не изменяет текст программы. Поэтому возможно использование макроса, переменной и функции с одним именем и в каждом случае можно изменять, когда нужно применить макрос (если за именем следует список аргументов), а когда - переменную или функцию (если список аргументов отстутствует).

Подобное двойственное использование одного имени может привести к осложнениям и его следует избегать, за исключением случаев, когда оба значения являются синонимами, то есть когда под одним именем определена функция и макрос и оба выполняют одинаковые действия. Можно рассматривать это имя как имя функции. Использование имени не для ссылки (например, для получения адреса) приведет к вызову функции, в то время как ссылка приведет к замене имени на значение макроса и в результате будет получен более эффективный но идентичный код. Например, используется функция с именем 'min' в том же исходном файле, где определен макрос с тем же именем. Если написать '&min' без списка аргументов, то это приведет к вызову функции. Если же написать 'min (x, bb)' со списком аргументов, то вместо этого будет произведена замена на значение соответствующего макроса. Если использовать конструкцию '(min) (a, bb)', где за именем 'min' не следует открывающаяся скобка, то будет произведен вызов функции 'min'.

Нельзя определять простой макрос и макрос с аргументами с одним именем.

В определении макроса с аргументами список аргументов должен следовать сразу после имени макроса без пробелов. Если после имени макроса стоит пробел, то макрос определяется без аргументов, а остальная часть строки становится значением макроса. Причиной этому является то, что довольно часто определяются макросы без аргументов. Определение макросов подобным образом позволяет выполнять такие операции как

#define FOO(x) - 1 / (x)

(где определяется макрос 'FOO', принимающий один аргумент и добавляет минус к числу, обратному аргументу) или

```
#define BAR (x) - 1 / (x)
```

(где определяется макрос 'BAR' без аргументов и имеющий постоянное значение '(x) - 1 / (x)').

Стрингификация

"Стрингификация" означает преобразование фрагмента кода в строковую константу, которая содержит текст этого фрагмента кода. Например, в результате стрингификации 'foo (z)' получается '"foo (z)"'.

В С препроцессоре, стрингификация является опцией, используемой при замене аргументов в макросе макроопределением. При появлении имени аргумента в теле макроопределения, символ '#' перед именем аргумента указывает на стрингификацию соответствующего аргумента при его подстановке в этом месте макроопределения. Этот же аргумент может быть заменен в другом месте макроопределения без его стрингификации, если перед именем аргумента нет символа '#'.

Вот пример макроопределения с использованием стрингификации:

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

Здесь аргумент 'EXP' заменяется один раз обычным образом (в конструкции 'if'), а другой - с использованием стрингификации (аргумет функции 'fprintf'). Конструкция 'do' и 'while (0)' является реализацией макроса 'WARN_IF (ARG);'.

Возможности сринги фикации ограничены до преобразования одного макро аргумента в одну строковую константу: не существует методов комбинирования аргумента с другим текстом и посследующей

стрингификации полученных данных. Хотя рассмотренный выше пример показывает как может быть достигнут подобный результат в стандартном ANSI C с использованием возможности объединения смежных строковых констант в одну. Препроцессор стрингифицирует реальное значение 'EXP' в отдельную строковую константу и в результате получается следующий текст:

```
do { if (x == 0) \
     fprintf (stderr, "Warning: " "x == 0" "\n"); } \
while (0)
```

но C++ компилятор обнаруживает три строковые константы, расположенные друг за другом и объединяет их в одну:

```
do { if (x == 0) \
     fprintf (stderr, "Warning: x == 0\n"); } \
while (0)
```

Стрингификация в C является не только заключением требуемого текста в кавычки. Необходимо помещать символ backslash перед каждым дополнительным символом кавычки, а также перед каждым символом backslash в символьной или строковой константе для получения строковой константы в стандарте C. Поэтому при стрингификации значения 'p = "foo\n";' в результате получится строка '"p = \"foo\\n\";"'. Однако символы backslash, не принадлежащие символьной или строковой константе, не дублируются: значение '\n' стрингифицируется в '"\n"'.

Пробелы (включая комментарии), находящиеся в тексте, обрабатываются в соответствии с установленными правилами. Все предшествующие и последующие пробелы игнорируются. Любые последовательности пробелов в середине текста в результате обработки заменяются на отдельный пробел.

Объединение

"Объединение" означает соединение двух строковых констант в одну. При работе с макросами, это означает объединение двух лексических единиц в одну более длинную. Один аргумент макроса может быть объединен с другим аргументом или с каким-либо текстом. Полученное значение может быть именем функции, переменной или типа, а также ключевым словом C. Оно даже может быть именем другого макроса.

При определении макроса, проверяется наличие операторов '##' в его теле. При вызове макроса и после подстановки аргументов все операторы '##', а также все пробелы рядом с ними (включая пробелы, принадлежащие аргументам) удаляются. В результате производится объединение синтаксических конструкций с обоих сторон оператора '##'.

Рассмотрим C программу, интерпретирующую указываемые команды. Для этого должна существовать таблица команд, возможно массив из структур, описанный следующим образом:

```
struct command
{
  char *name;
  void (*function) ();
};
struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

Более удобным будет не указывать имя каждой команды дважды: один раз в строковой константе, второй - в имени функции. Макрос, принимающий в качестве аргумента имя команды позволяет избежать это.

Строковая константа может быть создана с помощью стрингификации, а имя функции - путем объединения аргумента со строкой '_command'. Ниже показано как это сделать:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }
struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

Обычным объединением является объединение двух имен (или имени и какого либо числового значения) в одно. Также возможно объединение двух числовых значений (или числового значения и имени) в одно. Операторы, состоящие из нескольких символов (такие как '+='), также могут быть получены с помощью объединения. В некоторых случаях возможно объединение строковых констант. Однако, два текстовых значения, не образующих вместе правильной лексической конструкции, не могут быть объединены. Например, объединение с одной стороны символа 'x', а с другой - '+' является бессмысленным с точки зрения формирования лексических конструкций С. В стандарте ANSI указано, что подобный тип объединения не определен, хотя препроцессор GNU C их определяет. В данном случае он помещает вместе символы 'x' и '+' вместе без каких либо побочных эффектов.

Следует заметить, что препроцессор С преобразует все комментарии в пробелы перед обработкой макросов. Поэтому нельзя создать комментарий путем объединения '/' и '*' так как последовательность символов '/*' не является лексической конструкцией. Также можно использовать комментарии в макроопределениях после строки '##' или в объединяемых аргументах, так как сначала комментарии заменяются на пробелы, а при объединении эти пробелы игнорируются.

Удаление макросов

"Удалить" макрос означает отменить его определение. Это производится с помощью директивы '#undef', за которой следует имя макроса.

Как и определение, удаление макросов появляется в определенном месте исходного файла и вступает в силу с этого места. Например,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

заменяется на

```
x = 4;
x = FOO;
```

В этом примере значение 'FOO' должно быть лучше переменной или функцией, чем макросом, для получения после подстановки правильного C кода.

Директива '#undef' используется в такой же форме и для отмены макроопределений с аргументами или без них. Применение этой директивы к неопределенному макросу не дает никакого эффекта.

Переопределение макросов

"Переопределение" макроса означает определение (с помощью директивы '#include') имени, которое уже было определено как макрос.

Переопределение является простым, если новое определение явно идентично старому. Иногда не требуется специально выполнять простое переопределение, хотя оно производится автоматически, если подключаемый файл вкючается более одного раза, поэтому оно выполняется без какого-либо эффекта.

Нетривиальные переопределения рассматриваются как возможная ошибка, поэтому в таких случаях препроцессор выдает предупреждающее сообщение. Однако, это иногда помогает при изменении определения макроса во время предварительной компиляции. Появление предупреждающего сообщения можно запретить путем предварительного уничтожения макроса с помощью директивы '#undef'.

Для простого переопределения новое определение долно точно совпадать с предыдущим значением за исключением двух случаев:

В начале и в конце определения могут быть добавлены или удалены пробелы.

Пробелы можно изменять в середине определения (но не в середине строки). Однако они не могут быть полностью удалены, а также не могут быть вставлены туда, где их не было вообще.

Особенности использования макросов

В этом разделе рассматриваются некоторые специальные правила работы, связанные с макросами и макроподстановками, а также указываются отдельные случаи, которые следует иметь в виду.

Неправильно используемые конструкции

При вызове макроса с аргументами, они подставляются в тело макроса, а затем просматриваются полученные после подстановки данные вместе с оставшейся частью исходного файла на предмет дополнительных макро вызовов.

Возможно объединение макро вызова, исходящего частично от тела макроса и частично - от аргументов. Например,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

здесь строка 'call_with_1 (double)' будет заменена на '(2*(1))'.

Макроопределения не обязательно должны иметь закрывающиеся скобки. Путем использования не закрывающейся скобки в теле макроса возможно создание макро вызова, начинающегося в теле макроса и заканчивающегося вне его. Например,

```
#define strange(file) fprintf (file, "%s %d",
...
strange(stderr) p, 35)
```

В результате обработки этого странного примера получится строка 'fprintf (stderr, "%s %d", p, 35)'.

Нестандартная группировка арифметических выражений

Во большинстве примеров макроопределений, рассмотренных выше, каждое имя макроаргумента заключено в скобки. В дополнение к этому, другая пара скобок используется для заключения в них всего макроопределения. Далее описано, почему лучше всего следует писать макросы таким образом.

Допустим, существует следующее макроопределение:

```
#define ceil_div(x, y) (x + y - 1) / y
```

которое используется для деления с округлением. Затем предположим, что он используется следующим образом:

```
a = ceil_div (b & c, sizeof (int));
```

В результате эта строка заменяется на

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

которая не выполняет требуемой задачи. Правила приоритета операторов С позволяют написать следующую сроку:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

но требуется

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Если определить макрос следующим образом:

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

то будет получен желаемый результат.

Однако, нестандартная группировка может привести к другому результату. Рассмотрим выражение 'sizeof ceil_div(1, 2)'. Здесь используется выражение C, вычисляющее размер типа данных 'ceil_div(1, 2)', но в действительности производятся совсем иные действия. В данном случае указанная срока заменяется на следующую:

```
sizeof ((1) + (2) - 1) / (2)
```

Здесь определяется размер типа целого значения и делится пополам. Правила приоритета помещают операцию деления вне поля действия операции 'sizeof', в то время как должен определяться размер всего выражения.

Заключение в скобки всего макроопределения позволяет избежать подобных проблем. Далее дан правильный пример определения макроса 'ceil_div'.

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

Использование точки с запятой

Иногда требуется определять макросы, используемые в составных конструкциях. Рассмотрим следующий макрос, который использует указатель (аргумент 'p' указывает его местоположение):

```
#define SKIP_SPACES (p, limit)  \
{ register char *lim = (limit); \
  while (p != lim) {            \
    if (*p++ != ' ') {          \
      p--; break; }}}
```

Здесь последовательность backslash-newline используется для разбиения макроопределения на несколько строк, поскольку оно должно быть на одной строке.

Вызов этого макроса может выглядеть так: 'SKIP_SPACES (p, lim)'. Грубо говоря, при его вызове он заменяется на составную конструкцию,

которая является полностью законченной и нет необходимости в использовании точки с запятой для ее завершения. Но вызов этого макроса выглядит как вызов функции. Поэтому удобнее будет вызывать этот макрос следующим образом: 'SKIP_SPACES (p, lim);'

Но это может привести к некоторым трудностям при использовании его перед выражением 'else', так как точка с запятой является пустым выражением. Рассмотрим такой пример:

```
if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

Использование двух выражений (составной конструкции и пустого выражения) между условием 'if' и конструкцией 'else' создает неправильный С код.

Определение макроса 'SKIP_SPACES' может быть изменено для устранения этого недостатка с использованием конструкции 'do ... while'.

```
#define SKIP_SPACES (p, limit)     \
do { register char *lim = (limit); \
    while (p != lim) {             \
      if (*p++ != ' ') {           \
        p--; break; }}}            \
while (0)
```

Теперь макрос 'SKIP_SPACES (p, lim);' заменяется на

```
do {...} while (0);
```

что является одним выражением.

Удвоение побочных эффектов

Во многих С программах определяется макрос 'min' для вычисления минимума:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

При вызове этого макроса вместе с аргументом, содержащим побочный эффект, следующим образом:

```
next = min (x + y, foo (z));
```

он заменяется на строку

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

где значение 'x + y' подставляется вместо 'X', а 'foo (z)' - вместо 'Y'.

Функция 'foo' используется в этой конструкции только один раз, в то время как выражение 'foo (z)' используется дважды в макроподстановке. В результате функция 'foo' может быть вызвана дважды при выполнении выражения. Если в макросе имеются побочные эффекты или для вычисления значений аргументов требуется много времени, результат может быть неожиданным. В данном случае макрос 'min' является ненадежным.

Наилучшим решением этой проблемы является определение макроса 'min' таким образом, что значение 'foo (z)' будет вычисляться только один раз. В языке C нет стандартных средств для выполнения подобных задач, но с использованием расширений GNU C это может быть выполнено следующим образом:

```
#define min(X, Y)                \
({ typeof (X) __x = (X), __y = (Y);   \
   (__x < __y) ? __x : __y; })
```

Если не использовать расширения GNU C, то единственным решением будет осторожное применение макроса 'min'. Например, для вычисления значения 'foo (z)', можно сохранить его в переменной, а затем использовать ее значение при вызова макроса:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
...
{
  int tem = foo (z);
  next = min (x + y, tem);
```

```
}
```

(здесь предполагается, что функция 'foo' возвращает значение типа 'int').

Рекурсивные макросы

"Рекурсивные" макросы - это макросы, в определении которых используется имя самого макроса. Стандарт ANSI C не рассатривает рекурсивный вызов макроса как вызов. Он поступает на вывод препроцессора без изменений.

Рассмотрим пример:

```
#define foo (4 + foo)
```

где 'foo' также является переменной в программе.

Следуя обычным правилам, каждая ссылка на 'foo' заменяется на значение '(4 + foo)', затем это значение просматривается еще раз и заменяется на '(4 + (4 + foo))' и так далее, пока это не приведет к ошибке (memory full) препроцессора.

Однако, правило об использовании рекурсивных макросов завершит этот процесс после получения результата '(4 + foo)'. Поэтому этот макрос может использоваться для прибавления 4 к значению переменной 'foo'.

В большинстве случаев не следует опираться на эту возможность. При чтении исходных текстов может возникнуть путаница между тем, какое значение является переменной, а какое - вызовом макроса.

Также используется специальное правило для "косвенной" рекурсии. Здесь имеется в виду случай, когда макрос X заменяется на значение 'y', которое является макросом и заменяется на значение 'x'. В результате ссылка на макрос 'x' является косвенной и происходит от подстановки макроса 'x', таким образом, это является рекурсией и далее не обрабатывается. Поэтому после обработки

```
#define x (4 + y)
#define y (2 * x)
```

'x' заменяется на '(4 + (2 * x))'.

Но предположим, что 'y' используется где-либо еще и не в определении макроса 'x'. Поэтому использование значения 'x' в подстановке макроса 'y' не является рекурсией. Таким образом, производится подстановка. Однако, подстановка 'x' содержит ссылку на 'y', а это является косвенной рекурсией. В результате 'y' заменяется на '(2 * (4 + y))'.

Неизвестно где такие возможности могут быть использованы, но это определено стандартом ANSI C.

Отдельная подстановка макро аргументов

Ранее было объяснено, что макроподстановка, включая подставленные значения аргументов, заново просматривается на предмет наличия новых макро вызовов.

Что же происходит на самом деле, является довольно тонким моментом. Сначала значения каждого аргумента проверяются на наличие макро вызовов. Затем полученные значения подставляются в тело макроса и полученная макро подстановка проверяется еще раз на наличие новых макросов.

В результате значения макроаргументов проверяются дважды.

В большинстве случаев это не дает никакого эффекта. Если аргумент содержит какие-либо макро вызовы, то они обрабатываются при первом проходе. Полученное значение не содержит макро вызовов и при втором проходе оно не изменяется. Если же аргументы будут подставлены так, как они были указаны, то при втором проходе, в случае наличия макро вызовов, будет произведена макроподстановка.

Рекурсивный макрос один раз подставляется при первом проходе, а второй раз - при втором. Не подставляемые рекурсивные элементы при выполнении первого прохода отдельно помечаются и поэтому они не обрабатываются при втором.

Первый проход не выполняется, если аргумент образован путем

стрингификации или объединения. Поэтому

```
#define str(s) #s
#define foo 4
str (foo)
```

заменяется на '"foo"'.

При стрингификации и объединении аргумент используется в таком виде, в каком он был указан без последующего просмотра его значения. Этот же аргумент может быть просмотрен, если он указан где-либо еще без использования стрингификации или объединения.

```
#define str(s) #s lose(s)
#define foo 4
str (foo)
```

заменяется на '"foo" lose(4)'.

Возникает вопрос: для чего используется два прохода для просмотра макроса и почему бы не использовать один для повышения скорости работы препроцессора. В действительности, здесь есть некоторая разница и она может быть видна в трех отдельных случаях:

При однородных вызовах макросов.

При использовании макросов, вызывающих другие макросы, которые используют стрингификацию или объединение.

При использовании макросов, содержащих открытые запятые.

Макро вызовы называются "однородными", если аргумент этого макроса содержит вызов этого же макроса. Например, 'f' это макрос, принимающий один аргумент, а 'f (f (1))' является однородной парой вызовов макроса 'f'. Требуемая подстановка производится путем подстановки значения 'f (1)' и его замены на определение 'f'. Дополнительный проход приводит к желаемому результату. Без его выполнения значение 'f (1)' будет заменено как аргумент и во втором проходе оно не будет заменено, так как будет является рекурсивным элементом. Таким образом, применение второго прохода предотвращает

нежелательный побочный эффект правила о рекурсивных макросах.

Но применение второго прохода приводит к некоторым осложнениям в отдельных случаях при вызовах однородных макросов. Рассмотрим пример:

```
#define foo  a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))
bar(foo)
```

Требуется преобразовать значение 'bar(foo)' в '(1 + (foo))', которое затем должно быть преобразовано в '(1 + (a,b))'. Но вместо этого, 'bar (foo)' заменяется на 'lose(a,b)' что в результате приводит к ошибке, так как 'lose' принимает только один аргумент. В данном случае эта проблема решается путем использования скобок для предотвращения неоднородности арифметических операций:

```
#define foo (a,b)
#define bar(x) lose((x))
```

Проблема становится сложнее, если аргументы макроса не являются выражениями, например, когда они являются конструкциями. Тогда использование скобок неприменимо, так как это может привести к неправильному C коду:

```
#define foo { int a, b; ... }
```

В GNU C запятые можно закрыть с помощью '({...})', что преобразует составную конструкцию в выражение:

```
#define foo ({ int a, b; ... })
```

Или можно переписать макроопределение без использования таких запятых:

```
#define foo { int a; int b; ... }
```

Существует также еще один случай, когда применяется второй проход. Его можно использовать для подстановки аргумента с его последующей стрингификацией при использовании двухуровневых

макросов. Добавим макрос 'xstr' к рассмотренному выше примеру:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

Здесь значение 'xstr' заменяется на '"4"', а не на '"foo"'. Причиной этому служит то, что аргумент макроса 'xstr' заменяется при первом проходе (так как он не использует стрингификацию или объединение аргумента). В результате первого прохода формируется аргумент макроса 'str'. Он использует свой аргумент без предварительного просмотра, так как здесь используется стрингификация.

Зависимые макросы

"Зависимым" макросом называется макрос, тело которого содержит ссылку на другой макрос. Это довольно часто используется. Например,

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

Это не является определением макроса 'TABLESIZE' со значением '1020'. Директива '#define' для макроса 'TABLESIZE' использует в точности тело указанного макроса, в данном случае это 'BUFSIZE'.

Подстановка значения 'TABLESIZE' производится только при использовании этого макроса.

При изменении значения 'BUFSIZE' в каком-либо месте программы ее выполнение меняется. Макрос 'TABLESIZE', определенный как было описано выше, всегда заменяется с использованием значения макроса 'BUFSIZE':

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
```

Прикладная работа

1. Пример

Как с помощью макроса превратить объект в строку?

Вот пример такого макроса:

```
#define mIntToStr(nVar) (#nVar)
```

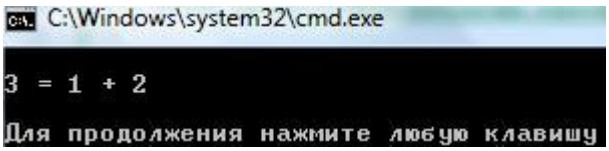
в этой макрофункции строковый оператор (так его иногда называют) # подействует на параметр nVar, в результате nVar будет охвачен кавычками. Код:

```
#include <stdio.h>

#define mIntToStr(nVar) (#nVar)

void main(void)
{
  printf("\n3 = %s\n\n", mIntToStr(1 + 2));
}
```

Получаем:



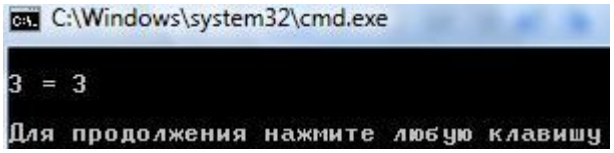
оператор # охватил параметр 1 + 2 кавычками, превратив его в строковый литерал. Если убрать # в макроопределении, то макрос mIntToStr сложит 1 + 2. Пример макроса:

```
#include <stdio.h>

#define mIntToStr(nVar) (nVar)

void main(void)
{
  printf("\n3 = %d\n\n", mIntToStr(1 + 2));
}
```

Получаем:



Как с помощью макроса объединять строки?

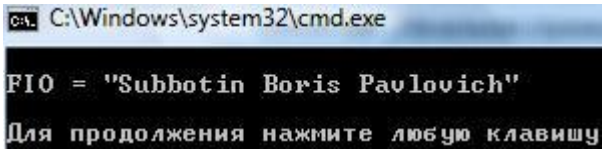
Строки в С можно объединить с помощью оператора ##. Пример макроса:

```
#include <stdio.h>

    #define mFIO(cQuote, szFamily, szName, szPatronymic) (cQuote##szFamily##\
    szName##szPatronymic##cQuote)

    void main(void)
    {
      char *szFIO = mFIO("\"","Subbotin ","Boris ","Pavlovich");
      printf("\nFIO = %s\n\n", szFIO);
    }
```

Получаем:



Заключение

В рамках курсовой работы были проанализированы а также создан макрос в C++ на основе средства записи макросов. Результатом курсовой работы, является созданный и подкорректированный макрос.

Независимо от используемой операционной системы и программных приложений пользователь часто выполняет одни и те же последовательности команд для многих рутинных задач. Вместо повторения последовательности команд каждый раз, когда необходимо выполнить какую-либо задачу, можно создать макрос, который будет выполнять эту последовательность. Макросы позволяют вводить одиночную команду, выполняющую ту же задачу, для реализации которой было бы необходимо вводить несколько команд вручную

В процессе выполнения курсовой работы было выявлено, что не все стандартные макросы возможно включить в собственный, их надо писать самому на языке C++.

Список литературы

1. А.Васильев, А.Андреев. VBA в Office 2000. Учебный курс. С-Пб.: "Питер", 2001
2. Биллиг В.А. Средства разработки VBA-программиста. Офисное программирование. Том 1. М.: Издательско-торговый дом "Русская Редакция", 2001.
3. Биллиг В.А. Мир объектов Excel 2000. М.: Издательско-торговый дом "Русская Редакция", 2001.
4. В.И.Король. Visual Basic 6.0, Visual Basic for Applications 6.0. Язык программирования. Справочник с примерами. М.: Издательство КУДИЦ, 2000.
5. В. Комягин, А. Коцюбинский, «Современный самоучитель работы на персональном компьютере», Москва 2003
6. О.А. Житкова, Т.И. Панфилова, «VBA в приложении к Excel, Word и Power Point» Москва 2006.
7. Что такое макрос? Режим доступа [http://www.codenet.ru/progr/vbasic/bit/Macros.php] по состоянию на 15.03.2011 г.
8. Макрос и макрорекордер Режим доступа [http://www.on-line-teaching.com/vba/] по состоянию на 15.03.2011 г.