Simplificando as listas encadeadas com a GList

Quem já estudou listas encadeadas em C/C++ e não resmungou do trabalho que dá usar este poderoso mas complicado recurso da linguagem que me atire a primeira pedra.

As listas encadeadas são recursos extramente importantes no desenvolvimento de aplicações em C e são usadas como uma espécie de "vetor" dinâmico, onde os elementos são ligados uns aos outros formando uma seqüencia.

Existem dois tipos básicos de lista encadeada: A lista encadeada simples e a lista duplamente encadeada. A diferença entre ambas é que a primeira só pode ser percorrida de frente pra trás, uma vez que cada elemento aponta apenas para o seu elemento seguinte; enquanto que a lista duplamente encadeada pode ser percorrida em ambos os sentidos uma vez que cada elemento possui referência ao seu antecessor e ao seu sucessor.

Cada um deste tipo de lista pode ser usado para se criar listas circulares, filas, pilhas, árvores etc. Mas isso está fora do escopo deste material e caso esteja interessado neste assunto você encontrará um farto material sobre isso apenas buscando no nosso bom amigo "google" (www.google.com.br) futuramente talvez eu venha a abordar esses assuntos em outro artigo, e até mesmo se você, caro leitor, se tiver conhecimento sobre o assunto e quiser contribuir estamos a sua disposição.

O desenho abaixo representa um elemento simples de uma lista encadeada

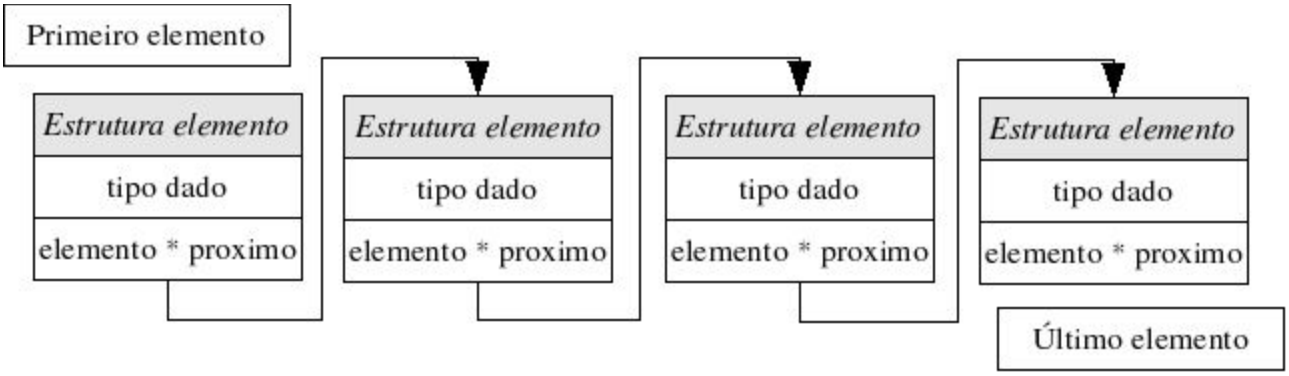

Figura1: lista encadeada simples

Note que cada elemento possui um ponteiro que aponta para o elemento seguinte da estrutura.

O objetivo deste artigo é mostrar como trabalhar com listas duplamente encadeadeadas, por elas terem uma quantidade de aplicações maior já que, como foi dito anteriormente, estas podem ser percorridas em dois sentidos. Mas se por algum motivo você precisar da lista encadeada simples isso poderá ser implementado facilmente com o tipo "GSList" que também está na biblioteca Glib.

Se você é daqueles que precisam de um bom motivo pra estudar, então vou lhe dar apenas um bom motivo para entender a lista encadeada: Ao fazer um programa com o Glade que use o objeto "gtkcombo" a lista de itens que são exibidos neste componente deve ser do tipo "GList" (que é uma lista duplamente encadeada!).

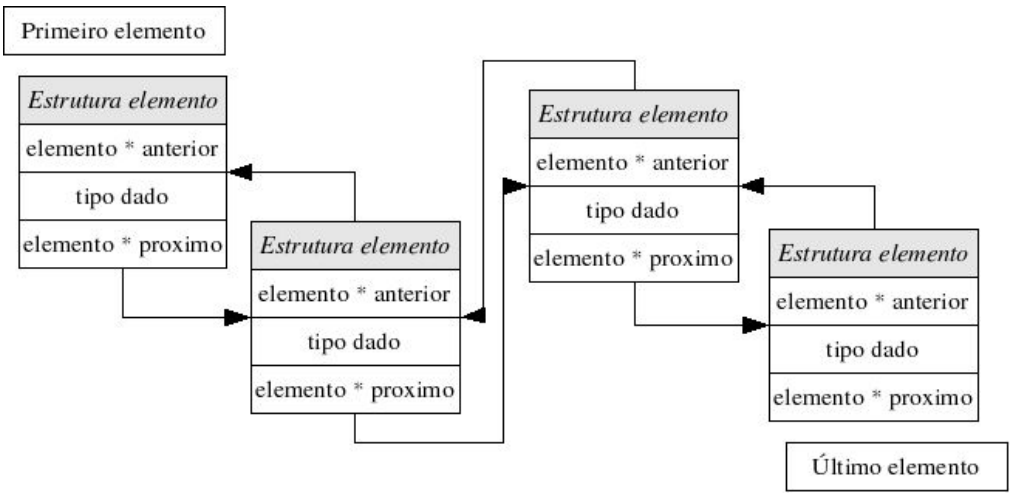

Figura 2: Lista duplamente encadeada

Observe no desenho da lista duplamente encadeada que cada elemento possui um ponteiro para o elemento seguinte, como na lista simples, mas possui também um outro ponteiro para o elemento anterior.

Listas duplamente encadeadas em C

Se ainda não ficou claro é bom notar que em uma lista encadeada cada elemento usa ponteiros para referenciar os outros elementos e como tudo deve ocorrer na dinamicamente precisamos usar as tradicionais funções "malloc" e "free" no C ANSI ou ainda as funções "new" e "delete" no C++.

Tipicamente uma lista encadeada para armazenar strings ficaria como segue:

```
struct s_strings {
  char *texto;
  s_strings *proxima;
  s_strings *anterior;
};
```

Observe que na prática nada mais temos do que uma estrutura simples contendo o dado principal que estamos querendo armazenar (nesse caso um ponteiro para "char*") e duas referências ao tipo "s_string" que apontarão para a estrutura anterior e para a seguinte.

Para o leitor que tem dificuldades em trabalhar com um único ponteiro e nunca trabalhou com listas, imagine lidar com todos esses ponteiros de uma só vez!

É claro que exercitando adquiri-se certa prática e habilidade, mas eu confesso que dei graças a Deus quando descobri o tipo GList da Glib. Portanto não vou me esticar mais mostrando como usar a lista mostrada acima, mas sim com a nossa Glib.

O tipo GList

A Glib implementa o tipo GList como sendo uma estrutura de lista duplamente encadeada nos moldes da estrutura já mostrada.

Olhando a documentação desta biblioteca (eu particularmente faço isso usando o programa "Devhelp" que é um navegador de documentação) podemos ver a estrutura que é exatamente o mostrado abaixo:

```
struct GList {
  gpointer data;
  GList *next;
```

```
  GList *prev;
};
```

Compare as duas estruturas e veja a semelhança. Observe ainda que o campo “data” é do tipo “gpointer”.

```
typedef void *gpointer;
```

Na realidade o gpointer é um ponteiro do tipo “void”, ou se você preferir um ponteiro “não-tipado” o que significa que o seu campo “data” poderá assumir qualquer tipo de dado, seja ele um dado simples como um “int” ou até mesmo uma complexa estrutura de dados.

Vale ainda ressaltar que em alguns momentos você verá o tipo “gconstpointer” que é representado como está abaixo:

```
typedef const void *gconstpointer;
```

Assim como “gpointer” este é um ponteiro para o tipo “void”, sendo que desta vez ele aponta para um dado constante. Isto geralmente é usado em protótipo de funções para que o dado passado a elas não seja alterado.

Para criarmos uma lista do tipo GList é necessário tão somente o código abaixo:

```
GList *minha_lista = NULL;
```

Observe que a lista deve ser iniciada como NULL para que não tenhamos problemas futuros. Assim ela é considerada vazia.

Funções Típicas para manipular a glist

Se considerarmos uma lista como um tipo de cadastro (um cadastro de endereços, por exemplo), as principais ações que são feitas nesse cadastro são: inclusão, remoção, classificação, procura e listagem. Veja então as funções que usamos pra isso.

Funções de inclusão

As funções a seguir são usadas para inclusão de dados na glist:

g_list_append()

Usada para adicionar um novo elemento no final da lista.

```
GList* g_list_append (GList *list, gpointer data);
```

g_list_prepend()

Usada para adicionar um novo elemento do início da lista.

```
Glist* g_list_prepend (GList *list, gpointer data);
```

g_list_insert()

Usada para adicionar um novo elemento em determinada posição.

```
Glist* g_list_insert (GList *list, gpointer data, gint position);
```

g_list_insert_sorted()

Usada para inserir elementos ordenadamente na lista.

```
Glist* g_list_insert_sorted (GList *list, gpointer data, GCompareFunc func);
```

Observe que a função g_list_insert_sorted() precisa receber o endereço de uma função de comparação, definida pelo programador previamente e seguindo algumas regras que vamos

comentar adiante.

Funções de remoção

As funções a seguir servem para remover elementos da nossa lista.

g_list_remove()

Use-a para remover um único elemento da lista. Para usar esta função basta apenas informar o nome da lista e o conteúdo do elemento a ser removido.

```
Glist* g_list_remove (GList *list, gconstpointer data);
```

É importante destacar que se existir dois elementos com o mesmo conteúdo, apenas o primeiro será removido. Se você desejar remover todos os elementos com este mesmo conteúdo use a função "g_list_remove_all()" que possui a mesma sintaxe.

g_list_free()

Esta função é usada para liberar a memória alocada pela sua estrutura e deve ser chamada explicitamente em seu programa quando tiver terminado com o uso da estrutura GList em questão.

```
void g_list_free (GList *list);
```

Outras funções

Seria uma perda de tempo ficar escrevendo cada função aqui e explicar o que já está explicado. Todas as funções da glib estão documentadas e podem ser consultadas diretamente no site oficial www.gtk.org , ou no site LiDN http://lidn.sourceforge.net que possui a documentação de praticamente todas as bibliotecas e ferramentas que você precisa e até mesmo localmente consultando os seus diretórios /usr/share/doc, /usr/doc e /usr/local/doc (Use um navegador de coumentação como o "Devhelp") para facilitar a sua vida.

Segue abaixo um breve resumo de mais algumas funções apenas para aguçar a sua curiosidade de ler a documentação.

- g_list_find() - Localiza um elemento com o dado especificado
- g_list_length() - Retorna o número de elementos na sua lista
- g_list_index() - Retorna a posição do elemento que contém o dado especificado
- g_list_foreach() - Executa uma determinada função para cada elemento da sua lista
- g_list_concat() - Junta duas listas
- g_list_sort() - Ordena uma lista de acordo com as características da função de ordenação especificada
- g_list_reverse() - Inverte a lista (o primeiro elemento torna-se o último, o segundo torna-se penúltimo etc)

A GList na prática

Chega de teoria e vamos ao que todos querem: Por em prática.

Pra começar vamos abrir um terminal e criar uma pasta para armazenar os arquivos deste programinha de teste.

Lembrando que para criarmos uma pasta, após ter aberto a janela de terminal basta emitir o comando:

```
$ mkdir nome-da-pasta [ENTER]
```

Eu particularmente gosto de armazenar estes testes dentro de uma pasta chamada "Projetos" então o que eu faço é entrar nesta pasta e depois criar a subpasta para o programa de teste:

```
$ cd Projetos [ENTER]
$ mkdir testeglist [ENTER]
```

Agora é só rodar o seu editor de textos preferido. Eu aconselho que este editor faça a numeraçã o de linhas, para facilitar o trabalho de depuração quando for necessário. (particularmente eu prefiro o Gedit).

```
$ gedit main.cc & [ENTER]
```

Com isso estamos rodando o gedit pronto para criar o arquivo principal do nosso teste e deixando o terminal livre para nobres tarefas, como rodar o gcc por exemplo.

Vamos começar digitando o seguinte código:

```
/*
 *  Teste da Glist
 *  para compilar use:
 *    gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
 */

//Inclusao da biblioteca glib
//isso e importante para usar o tipo GList que é o foco do nosso estudo
#include <glib.h>

//Criacao da Lista encadeada
GList * MinhaLista;

//Funcao principal do nosso programa de teste
int main (char argc, char **argv) {

  g_print("Teste da Glist\n");

  return 0;

}
```

Até aqui, se compilarmos o nosso programa ele não deverá acusar qualquer erro, mas se for executado, também não deverá fazer nada de interessante.

Para não termos que nos estressar com os métodos de entrada do C (aqueles “scanf” da vida) vamos aproveitar o vetor de paramêtros do nosso programa e entrar com os itens da lista via linha de comandos.

Para isso inclua o código abaixo a partir da linha 14 (onde está o “return 0”).

```
  gint i = 0; //Variável inteira de uso geral
  MinhaLista = NULL; //É importantíssimo que a GList seja iniciada com NULL
```

Veja que nós somente criamos uma variável do tipo inteiro que será usada para diversos fins e o mais importante que é iniciar a GList com o valor NULL. Lembre-se que uma lista encadeada é um ponteiro e todos os ponteiros devem ser sempre iniciados para não termos dor de cabeça futuramente.

Inclusão de elementos na lista

Agora inclua o código seguinte logo acima do “return 0;” (linha 16).

```
  if (argc > 1) { //Verifica se foi passado algum argumento
    for (i = 1 ; i < argc ; i++ ) {//Passa por todos os elementos da lista
      //Adiciona cada um dos elementos no final da lista
      MinhaLista = g_list_append(MinhaLista,(gpointer) argv[i]);
    } //for i
  } //if argc
  g_list_free(MinhaLista); //Libera o espaço alocado pela "MinhaLista"
```

Aqui a coisa começou a ficar interessante: primeiramente estamos verificando se foi passado algum parâmetro para o nosso programa (verificando o valor de "argc"). Se houver pelo menos um item (maior que 1, pois o nome do executável é o primeiro parâmetro) então começamos um loop do primeiro até o último e adicionamos cada um deles na nossa GList usando a função "append" que inclui um elemento no final da lista.

Observe que os nossos items são do tipo "char*", mas esta função requer um ponteiro "não-tipado" ("void *" ou "gpointer"), daí o motivo do "casting" que foi feito.

Outra coisa a frisar é a função "g_list_free()" antes de encerrar o programa. Isso vai liberar a memória usada pela nossa GList sem precisarmos nos preocupar de remover itempor item na lista.

Quantos itens há na lista?

Se compilarmos o nosso programa e o rodarmos ele não deverá acusar nenhum erro, mas nós ainda não podemos saber o que está acontecendo com ele, pois apesar de estarmos inserindo itens na lista nós não temos nenhuma informação a respeito disso.

Insira a partir da linha 21 ("} //if argc") o seguinte código:

```
    //Mostra o total de elementos na glist
    g_print("[Total de elementos: %d]\n",g_list_length(MinhaLista));
```

Com essa linha logo após o laço "for" nós poderemos ver a quantidade de elementos que foram inseridos na lista.

O comando "g_print" é o responsável por exibir na tela a mensagem informando o tamanho (sua sintaxe é a mesma do já conhecido "printf" do C) e o comando "g_list_length" retorna a quantidade de elementos na lista, precisando apenas receber a lista como parâmetro.

Agora compile o programa e rode. A saída deverá ser algo como o mostrado abaixo:

```
$ gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
$ ./testeglist GTK-Brasil www.gtk.cjb.net
Teste da GList
[Total de elementos: 2]
```

Emitindo um aviso em caso de falha

Observe que executamos o testeglist com dois argumentos e ambos foram adicionados à lista. Experimente rodar o programa sem argumentos e você verá que não teremos nenhuma mensagem informando o que está acontencendo não é uma boa pratica de programação deixar o usuário sem informação portanto vamos substituir a linha 23 (" } //if argc") pelo código a seguir:

```
  } else g_warning("Nenhum item foi passado para lista. A lista esta
vazia");
```

Se o programa for compilado e executado agora, sem parâmetros teremos a seguinte saída:

```
$ gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
$ ./testeglist
Teste da GList

** (process:4650): WARNING **: Nenhum item foi passado para lista. A lista
esta vazia
```

Veja que usamos o "g_warning()" ao invés da "g_print" para apresentar a mensagem de aviso. Isso assegura que esta mensagem seja passada para o console de erros (/dev/stderr ou outro que seja definido).

É interessante nós fazermos isso nos nossos programas para que póssamos filtrar as mensagens de erro quando necessário. Se você se interessou por este recurso veja o manual da glib e procure a

respeito das funções (“g_print()”, “g_message()”, “g_warning”, “g_critical()” e “g_error()”).

Exibindo o que há na lista

Até aqui nosso programa já armazena elementos na lista e informa quanto foram armazenados, mas ainda não sabemos quais foram armazenados. Não existe uma função que apresente todo o conteúdo da lista, mas para resolver esse problema e mostrar o conteúdo da nossa lista usaremos a função “g_list_foreach()” que executa uma função criada pelo usuário para cada elemento da lista. Mas antes que isso seja possível precisamos definir o protótipo desta função de exibição que deverá se parecer com o protótipo abaixo:

```
void (*GFunc) (gpointer data, gpointer user_data);
```

Onde:

- (*GFunc) - é o nome da função,
- data - é o elemento que será passado pela função “g_list_foreach()” (no nosso caso será um “gchar*”)
- user_data - pode ser um vetor de parâmetros extras para configurar a sua função.

Começaremos inserindo o protótipo a seguir antes da função “main” (linha 11):

```
void mostra (gpointer item, gpointer meusdados); //Mostra o item da lista
```

e o corpo da função listado abaixo no final do arquivo, após o “}” que fecha a nossa função “main” (linha 28):

```
void mostra (gpointer item, gpointer meusdados){ //Mostra o item da lista
  gchar *texto = (gchar *) item;

  //Executa o que foi prometido ;-)
  g_print(" * %s\n",texto);

}
```

A primeira linha útil desta função apenas cria uma variável onde o texto de cada elemento da nossa lista ficará armazenado e atribui a ela o item atual já convertido para “gchar *”.

Em seguida ele mostra esse texto usando a função “g_print()”.

Não é preciso dizer que apesar de termos criado a função ela ainda não é chamada em ponto algum do nosso programa, portanto antes de compilarmos o nosso programinha nós vamos até a linha 24 do nosso programa (entre o *g_print(“[Total...”)* e o *} else g_warning()*) e acrescentaremos a seguinte linha:

```
    g_list_foreach(MinhaLista,mostra,"");
```

Como já dito antes a função “g_list_foreach” executa uma determinada função para cada elemento contido na lista. Nós a usamos para executar a função “mostra” que haviamos criado e pronto. Observe que no final foi passado “” para o parâmetro “meusdados” que pra esse caso não tem utilidade alguma, mas deve ser passado, nós veremos um exemplo prático de uso deste parâmetro mais adiante quando vermos a remoção de elementos.

Finalmente compile e rode o programa.

```
$ gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
$ ./testeglist Ana Jose Maria Teresa
Teste da GList
[Total de elementos: 4]
 * Ana
 * Jose
 * Maria
 * Teresa
```

Ordenando a lista

Para ordenar a lista nós usamos a função “g_list_sort()” ou a “função g_list_sort_with_data()”. Ambas precisam de uma função auxiliar que determina o critério de ordenação. A diferença que existe entre elas é que a segunda aceita parâmetros adicionais que serão enviados para a sua função de ordenação.

Veja abaixo os seus protótipos:

```
GList* g_list_sort (GList *list, GCompareFunc compare_func);
```

e

```
GList* g_list_sort_with_data (GList *list, GCompareDataFunc compare_func,
gpointer user_data);
```

Ambas retornam um ponteiro para o início da lista já ordenada e seus parâmetros são:

- list – A lista a ser ordenada
- compare_func – A função que define o critério de ordenação
- user_data – Dados que serão passados a função “compare_func” (só na função “g_list_sort_with_data”)

O protótipo da função “compare_func” para a a função “g_list_sort” é o que segue:

```
gint (*GcompareFunc) (gconstpointer a, gconstpointer b);
```

E para a função “g_list_sort_with_data” é:

```
gint (*GcompareDataFunc) (gconstpointer a, gconstpointer b, gpointer
user_data);
```

Ambas recebem dois itens que serão comparados (enviados pelas funções “g_list_sort” e “g_list_sort_with_data”) e a última recebe ainda algum argumento que seja necessário passar para sua função.

Ao criar estas funções você deve considerar que os valores de retorno devem seguir a regra:

- Valor negativo se a < b ;
- 0 se a = b ;
- valor positivo se a > b.

Ex.:

Considere uma lista contendo os itens: M R B M

Ao comparar “M” com “R” deve ser retornado valor positivo (5), pois “M” é menor que “R”

Ao comparar “M” com “B” deve ser retornado valor negativo (-11), pois “M” é maior que “B”

Ao comparar “M” com “M” deve ser retornado valor 0 (zero) , pois “M” é igual a “M”

Pra quem ainda não entendeu o porque disso, considere a tabela ASCII onde cada caracter corresponde a um número de 0 a 255. “M” vale 77, “R” vale 82 e “B” vale 66. O que o computador faz é subtrair os valores ASC do primeiro com o segundo e retornar a diferença. Você pode ver essa tabela com o programinha abaixo:

```
#include <glib.h>
int main() {
  int i;
  for (i = 33 ; i < 127 ; i++)
    g_print("%3d = [%c]\n",i,i);
```

```
  return 0;
}
```

O que este programinha fará é listar toda a tabela a partir do código 33 (“!”) até o código 126 (“~”).

Pra compilá-lo você usará uma linha similar a que estamos usando para compilar e rodar o nosso teste da glist:

```
$ gcc ascii.c -o ascii `pkg-config --cflags --libs glib-2.0`
$ ./ascii
```

Nós usaremos a função “g_list_sort()” e para isso vamos definir o protótipo da nossa função de comparação logo abaixo do protótipo da função “mostra()” (linha 12):

```
gint compara(gconstpointer a,gconstpointer b);
```

e o corpo da função será digitado lá no final do arquivo, após a função mostra (linha 38).

```
gint compara(gconstpointer a,gconstpointer b) { //Compara duas strings
  return g_ascii_strncasecmp( (const gchar *) a , (const gchar *) b , 5) ;
}
```

Observe que o protótipo da função corresponde ao da função “GcompareFunc” e a comparação é feita na realidade pela função “g_ascii_strncasecmp()” que é usada para comparar strings. Esta função é a substituta da função “g_strncase()” que tornou-se obsoleta e por isso não deve ser usada em novos códigos.

Consultando no manual, por estas funções vemos que elas recebem dois ponteiros para “const char*” e o número de caracteres que serão usados para comparação, no nosso caso serão apenas os 5 primeiros.

Agora que já temos a função definida só precisamos chamá-la junto com o “g_list_sort()” para que tenhamos a nossa lista ordenada.

Vamos incluir o código abaixo logo acima do “g_list_for_each()” que usamos anteriormente para exibir o conteúdo da lista (linha 25).

```
    MinhaLista = g_list_sort(MinhaLista,compara); //Ordena a lista

    g_print("\nLista Ordenada\n");
```

Com isso estamos ordenando a lista e em seguida exibindo-na na tela.

Agora compile e rode o programa para ver o resultado.

```
$ gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
$ ./testeglist Linux GNU GPL Debian Glib Kernel

Teste da GList
[Total de elementos: 6]


Lista Ordenada
 * Debian
 * Glib
 * GNU
 * GPL
 * Kernel
 * Linux
```

Observe que a lista foi corretamente ordenada e mesmo tendo indicado que a ordenação ocorre até o 5º caracter de cada elemento mesmo os itens com menor tamanho foram ordenados certos. Saiba ainda que se tivesse algum elemento com tamanho maior ele seria ordenado normalmente também.

Vale ainda uma nota a respeito da função “g_list_insert_sorted” que pode se aproveitar da nossa função “compara” para inserir os itens já ordenados na nossa lista. Assim poderiamos reduzir a quantidade de comandos no nosso programa substituindo o “append” seguido do “sort” por apenas o “insert_sorted”.

Se quiser experimentar (eu recomendo fazer isso, para fixar) então comente a linha 21 ("g_list_append") e insira abaixo dela a linha a seguir, depois comente a linha do comando "g_list_sort" (linha 26)

```
      MinhaLista = g_list_insert_sorted(MinhaLista,(gpointer) argv[i],
compara); //Insere ordenando
```

Com isso todo elemento que for inserido à lista entrará na posição adequada.

Invertendo a lista

Para invertermos a lista existe uma maneira bem simples que é usando a função "g_list_reverse()" como mostrado abaixo.

```
    MinhaLista = g_list_reverse(MinhaLista);     //Inverte a lista
```

Inclua isso após a função de ordenação (linha 26) e veja o resultado. Mais simples que isso é pedir demais! É claro que pra esse caso não haveria essa necessidade, pois se queremos apenas classificar em ordem descendente (de Z para A) poderiamos incluir um sinal de menos "-" no comando "return" da função "compara" como abaixo:

```
return -g_ascii_strncasecmp( (const gchar *) a , (const gchar *) b , 5) ;
```

A função "g_list_reverse" é dedicada aos casos onde você precisa inverter o conteúdo da lista independente da ordem de classificação. Por exemplo a listagem vinda de um arquivo com histórico de endereços num navegador web, onde você deseja exibir do último (acessado recentemente) para o primeiro (mais antigo).

Removendo um elemento da lista

Para remover elementos da lista podemos usar as funções "g_list_remove()" ou "g_list_remove_all ()". A diferença entre elas é que havendo dois ou mais elementos com o mesmo conteúdo a primeira função removerá apenas o primeiro elemento da lista e a segunda função removerá todos.

Por exemplo, suponha uma lista contendo: {A B C D E F A B C D E A B C D}

Se usarmos a função "g_list_remove" para remover o elemento "A" a lista passará a ser: {B C D E F A B C D E A B C D}, enquanto que se usarmos a "g_list_remove_all" a lista passará a ser: {B C D E F B C D E B C D}.

O protótipo das funções é o que segue abaixo:

```
GList* g_list_remove (GList *list, gconstpointer data);
```

e

```
GList* g_list_remove_all (GList *list, gconstpointer data);
```

Ambas precisam receber a lista a ser alterada e o conteúdo do elemento a ser removido. Veja um exemplo simples abaixo:

```
    MinhaLista = g_list_remove(MinhaLista,(gconstpointer)argv[1]);
```

Neste exemplo estamos removendo o ultimo elemento inserido na lista.

Mas vale destacar aqui um problema que encontramos com frequencia. Insira esta linha logo abaixo do "} //for i" (linha 24)

Compile e rode o programa. Observe que o ultimo elemento inserido ("Linux") foi removido. Certo?!

```
$ gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
$ ./testeglist Linux GNU GPL Debian Glib Kernel
```

```
Teste da GList
[Total de elementos: 5]


Lista Ordenada
 * Debian
 * Glib
 * GNU
 * GPL
 * Kernel
```

Agora substitua a variável “argv[1]” pelo próprio valor, ou seja “Linux” como abaixo:

```
    MinhaLista = g_list_remove(MinhaLista,(gconstpointer)”Linux”);
```

Compile e rode. Veja que a lista ficou intacta, ou seja o “g_list_remove” por algum motivo que parece inexplicável não removeu o elemento.

Por que isso aconteceu?! Simples! Se você olhar como nós inserimos os valores na lista, verá que o seu elemento “data” não contém as strings em si, mas apenas um ponteiro indicando onde está o “argv[]” correspondente. Logo, não adianta indicar o texto, mas sim o endereço.

Uma forma de resolver esse problema é usar o foreach para percorrer a lista e em cada elemento ele pega o item e compara o valor desejado. Apesar de estranho essa forma é a mais comum de se usar, pois você geralmente usará uma estrutura como por exemplo uma “struct” com vários dados, ao invés de uma simples string.

Vejamos a solução:

Acrescente esse protótipo junto dos demais:

```
void remover(gpointer item,gpointer valor); //Remove item
```

No final do arquivo inclua o corpo da função:

```
void remover(gpointer item,gpointer valor) { //Remove item
  if (compara(item,valor) == 0)
      MinhaLista = g_list_remove(MinhaLista,item);
}
```

Agora seja feliz! Inclua uma chamada para o foreach no lugar daquele g_list_remove que não funcionou (linha 25).

```
    g_list_foreach(MinhaLista,remover,"Linux");
```

Veja que nós aproveitamos a função “compara” que já haviamos criado. Isso nos garante que se mudarmos o tipo de dado principal da nossa lista a remoção estará atualizada também. O seu funcionamento depende do “foreach” que chama a nossa função “remove” passando a ela o campo “data” do elemento atual e o valor que queremos excluir. Ao fazer isso a função compara se ambos são identicos (retorna 0 se forem iguais). Se realmente for o mesmo então o item é removido.

Você poderá experimentar agora que independente de passar uma constante como “Kernel”, “Linux” etc. ou uma variável como “argv[1]” o item será localizado e removido.

Esta construção só tem um problema: Como o “foreach” é executado em todos os elementos, independentemente se você usar a função “remove” ou a função “remove_all” todos os elementos com aquele texto serão removidos. Para resolver esse problema, se isso for inconveniente para a sua aplicação é possível fazermos um loop manualmente com um laço while e após encontramos o item a ser removido quebramos esse laço com o “break” do “C/C++”.

Compile e rode o programa passando o primeiro argumento repetido várias vezes:

```
$ gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
$ ./testeglist Linux GNU GPL Debian Glib Kernel Linux Linux
Teste da Glist
[Total de elementos: 5]
```

```
Lista Ordenada
 * Debian
 * Glib
 * GNU
 * GPL
 * Kernel
```

Observe que o argumento “Linux” que foi passado três vezes foi completamente extinto da lista. Veja a solução para este problema.

Nós vamos criar uma função seguindo o que foi explanado acima. O protótipo abaixo mostra que a nossa função é bem similar ao “g_list_remove” da Glib, mudando apenas o valor que na função original é um “constpointer” e esta nossa função eu usei o tipo mais adequado ao programa em questão que é um “const gchar*”, além do retorno que o normal era ponteiro para a nova lista e nessa função personalizada ela retorna “void”. Você poderá incluí-lo junto aos demais protótipos de funções que já temos.

```
void g_list_remove_em(GList *lista , const gchar *valor); //Remove um item
```

O corpo da função é este e pode ser incluso no final do nosso arquivo:

```
void g_list_remove_em(GList *lista , const gchar *valor){ //Remove um item
  lista = g_list_first(lista);
  do
    if (compara(lista->data,valor) == 0) {
      lista = g_list_remove(lista,lista->data);
      break;
    }
  while (lista = g_list_next(lista));
}
```

E a forma de chamada é esta e você poderá incluí-lo no lugar daquele foreach que usamos para remover elementos, você poderá experimentar substituir o “argv[1]” por um elemento constante qualquer e ver o resultado:

```
    g_list_remove_em(MinhaLista,”Linux”);
```

Vejam que não é uma função complexa. Na chamada nós passamos a lista e o texto a ser removido. A primeira coisa que a função faz é garantir que estamo no início da lista (função “g_list_first”) e fazer um loop enquanto há um próximo elemento (função “g_list_next”). Durante a iteração deste laço nós verificamos se o componente “data” da nossa lista é o mesmo do valor passado. Se for nós o removemos com o “g_list_remove” e quebramos (break) o laço de forma a garantir que mesmo existindo outro elemento com esse mesmo texto ele não será removido.

Compilando e rodando o programa temos o seguinte resultado:

```
$ gcc main.c -o testeglist `pkg-config --cflags --libs glib-2.0`
$ ./testeglist Linux GNU GPL Debian Glib Kernel Linux Linux
Teste da Glist
[Total de elementos: 7]

Lista Ordenada
 * Debian
 * Glib
 * GNU
 * GPL
 * Kernel
 * Linux
 * Linux
```

Observe agora que mesmo tendo passado o parâmetro “Linux” três vezes apenas o primeiro foi removido.